C7

Page 3 File Index Fri Jan 04 14:40:00 2008
Page 4 File Index Fri Jan 04 14:40:00 2008

```
%/*
%** Copyright 1997,1998 EMC Corporation
%*/

/*
**  Leading % causes rpcgen to pass a line directly thought to the output,
**  ie restore_engine.h in this case.  This allows the .h to make a little
**  more sense and be properly documented.
*/

%/*
% * restore_engine.x : EDM Restore Engine C/S communication module
% *
% * Mission Statement:  This is an RPCGEN file which defines the RPC interface
% *                     between the Restore Engine server (which resides on
% *                     the EDM server) and  the backup client callers of its
% *                     functions.  This defines the RPC level calls that a
% *                     "caller" can make and the "service" will respond to.
% *
% * Primary Data Acted On: This defines the data that will flow over the wire.
% *                        The RPC mechanism will take care of data
                                                              marshalling
% *
% *
% * Compile-Time Options:
% *                     This acutally gets run through RPCGEN not compiled.
                                                                         It
% *                      must be run through with the -h flag to create a
% *                      header, the -m flag to create the service side
% *                      routines, the -l flag to create the client side
% *                      routines, and the -c flag to create the common data
% *                      marshalling routines.
% *
% * Basic idea here:
% *                     Define the RPC level interfaces to the Restore Engine
% *                     and all data types that will be passed via RPC.
%*/

/* for sharing of STRING(x) and OPAQUE(x) */
#define  IN_DOTX
#include <restore/restoreRPC.h>

%#include <restore/dispatch_daemon.h>

/*********************************************************************
 Constant Definitions
*********************************************************************/

/*********************************************************************
 Enum Definitions
*********************************************************************/

/*********************************************************************
 Typedef Definitions
*********************************************************************/

typedef int      RE_errno_ty;

/*********************************************************************
 Data Structure Definitions
*********************************************************************/

/* Structure to start every RPC request and response - for debug purposes */
        struct RE_rpc_objID
        {
               unsigned long   rpc_type;
```

```
                                        /* RPC Object ID (ie, rpc #) */
               RSTRPC_time_ty  time;   /* creation time */
               long            len;
                                      /* Length of structure, version num? */
        };

        struct RE_null_args {
               RE_rpc_objID RPCobjID;
        };

        struct RE_status_result {
               RE_rpc_objID RPCobjID;
               RE_errno_ty status;
        };

        struct RE_boolean_result {
               RE_rpc_objID    RPCobjID;
               RE_errno_ty     status;
               RSTRPC_bool     boolResult;
        };

        union RE_restorable_obj switch (RSTRPC_ObjectLevel objLevel)
        {
        case RSTRPC_tlo_type:
               RSTRPC_top_level_obj              *tloInfo;
        default:  /* anything else means NOT tlo -- i.e. container or leaf */
               RSTRPC_user_restorable_object    *uroInfo;
        } ;

const MAX_CHOICE_TEXT=80;

struct Choices {
    bool        isset;
    string       ctext<>;
    Choices      *nextchoice;
};

/* Question types */
const QTYPE_BOOL =       1;
const QTYPE_RAD =        2;
const QTYPE_MULTI =      4;
const QTYPE_STR =        8;
const QTYPE_YESNO =      16;
const QTYPE_INT =        32;

struct Question {
    int          qnum;
    int          qtype;
    int          maxlen;
    int          minlen;
    int          numchoices;
    string       invalidchars<>;
    string       headertxt<>;
    string       qtext<>;
    Choices      *choices;
};

struct Answer {
    int          qnum;
    string       ctext<>;
    Answer       *nextanswer;
};

struct AnswerList {
    int          numanswers;
    Answer       *firstanswer;
```

```
};


/* structures for input and output of re_initialize rpc call:  */

        struct RE_initialize_args {
                RE_rpc_objID RPCobjID;
                string username<>;
        };

/* structures for input and output of get_source_hosts and
 * get_destination_hosts rpc calls:
 */
        struct RE_get_hosts_args {
                RE_rpc_objID    RPCobjID;
                string          hostname<>;
                                                /* only for get_source_hosts */
                short           maxEntries;
                long            cookie;
        };

        struct RE_get_hosts_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                short           numEntries;     /* redundant but useful ? */
                long            cookie;
                RSTRPC_name_list        *hosts;  /* link to first hostname */
        };

/* structure for single character string argument */
        struct RE_string_args {
                RE_rpc_objID    RPCobjID;
                string  name<>;
        };

/* structure for GetHostPlatformType results */
        struct RE_get_host_platform_type_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                int             ptype;
        };

/* structures for input and output of submit RPCs
 */
        struct RE_submit_args {
                RE_rpc_objID    RPCobjID;
                string          hostname<>;
                string          directory<>;
                int             overwritePolicy;
                bool            inPlace;
                int             transport;
                int             submitObjectID;
                int             socketPort;
                string          socketClientName<>;
                string                  mapFile_env<>;
        };

        struct RE_get_submit_results_args {
                RE_rpc_objID    RPCobjID;
                bool            interrupt;
        };

        struct RE_get_submit_results_output {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                int             submitObjectID;
```

```
                                                /* handle for submit object */
                u_long          objectsDone;
        };

/* structures for input of Start RPC
 */
        struct RE_start_args {
                RE_rpc_objID    RPCobjID;
                int             submitObjectID;
                                                /* handle for submit object */
        };

/* structures for input and output of get_restore_feedback RPC
 */
        struct RE_get_restore_feedback_args {
                RE_rpc_objID    RPCobjID;
                bool            quit_restore;   /* flag to request cancel */
        };

        struct RE_Notification {
                int             msgtype;
                int             sourcemodule;
                int             level;
                int             msglen;
                string          msgtext<>;
                RE_Notification         *next;
        };

        struct RE_get_restore_feedback_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
               EDMStats         rstStats;
               RE_Notification  *notify;
        };

/* structure for output of get_question RPC
 */
        struct RE_get_question_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                Question        *query;
        };

/* structure for input of set_user_answer RPC
 */
        struct RE_set_user_answer_args {
                RE_rpc_objID    RPCobjID;
                AnswerList      answers;
        };

/* structures for input and output of get_top_level_objects RPC
 */
        struct RE_get_top_level_objects_args {
                RE_rpc_objID    RPCobjID;
                string          sourceHost<>;
                short           maxEntries;
                long            cookie;
        };

        struct RE_get_top_level_objects_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_tlo_list *topLevelObjs;  /* linked list */
                short           numEntries;
                long            cookie;
        };
```

```
/* structures for input and output of get_workitem_templates rpc call:
 */
        struct RE_get_top_level_templates_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_top_level_obj *topLevelObj;
                short           maxEntries;
                long            cookie;
        };

        struct RE_get_top_level_templates_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                short           numEntries;     /* redundant but useful ? */
                long            cookie;
                RSTRPC_name_list *templates;    /* link to first template */
        };

/* structure for input of does_alternate_exist rpc call:
 */
        struct RE_does_alternate_exist_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_top_level_obj *topLevelObj;
                string          templateName<>;
        };

/* structures for input and output of get_restorable_objects RPC's:
 */
        struct RE_get_restorable_objects_start_args {
                RE_rpc_objID    RPCobjID;
                RE_restorable_obj *parentObj;
                long            cookie;
                short           maxEntries;
                bool            allowBadFiles;
        };

        struct RE_get_restorable_objects_start_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
        };

        struct RE_get_restorable_objects_output_args {
                RE_rpc_objID    RPCobjID;
                short           maxEntries;
        };

        struct RE_get_restorable_objects_output_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_uro_list *childrenObjs;  /* linked list */
                long            numEntries;
                long            cookie;
        };

/* structures for input and output of find_restorable_objects RPC's:
 */
        struct RE_search_criteria {
                string  startDirectory<256>; /* Dir to start searching */
                bool     descendDirectory; /* Flag to descend into subdirs */
                string  searchString<128>;   /* String to search for */
                bool     excludeString;    /* Flag to include or exclude */
                RSTRPC_enum_ty  typeOfFile;
                                           /* Types of files to search for */
                string  owner<64>;           /* Specific owner of files */
                bool     excludeOwner;     /* Flag to exclude owner */
                string  group<64>;           /* Specific group of files */
```

```
                bool     excludeGroup;    /* Flag to exclude group */
                RSTRPC_u_hyper sizeInBytes;
                                          /* Specific size of files to find */
                RSTRPC_enum_ty  sizeMatch;
                                        /* type of matching to do for size */
                RSTRPC_time_ty  startTime;
                                               /* First backup date to use */
                RSTRPC_time_ty  endTime;        /* Last backup date to use */
        };

        struct RE_find_restorable_objects_args {
                RE_rpc_objID    RPCobjID;
                RE_search_criteria *searchCriteria;
        };

        struct RE_find_restorable_objects_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
        };

        struct RE_get_find_results_args {
                RE_rpc_objID    RPCobjID;
                bool            interrupt;      /* flag to request cancel */
                short           maxEntries;
                long            cookie;
        };

        struct RE_get_find_results_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_found_obj_list   *foundObjs;     /* linked list */
                long            numEntries;
                long            cookie;
        };

/* structures for input and output of mark_object RPC's:
 */
        struct RE_mark_object_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_user_restorable_object *thisObj;
                RSTRPC_time_ty  backupTime;
                bool            allowBadFiles;
                bool            descend;
        };

        struct RE_mark_object_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
        };

        struct RE_get_mark_results_args {
                RE_rpc_objID    RPCobjID;
                bool            interrupt;      /* flag to request cancel */
        };

        struct RE_get_mark_results_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                u_long          badFileCount;
                u_long          permDenyFileCount;
                u_long          dirMarkCount;
                u_long          fileMarkCount;
                u_long          otherMarkCount;
        };

/* structures for input and output of unmark_object RPC's:
 */
        struct RE_unmark_object_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_user_restorable_object *thisObj;
                RSTRPC_time_ty  backupTime;
                bool            badFilesOnly;
                bool            descend;
        };

        struct RE_get_unmark_results_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                u_long          badFileCount;
                u_long          dirMarkCount;
                u_long          fileMarkCount;
                u_long          otherMarkCount;
        };

/* structure for output of get_marked_total_size RPC:
 */
        struct RE_get_marked_total_size_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_u_hyper  total;
        };

/* structure for output of get_current_template RPC:
 */
        struct RE_get_current_template_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                string          templateName<>;
                RSTRPC_bool     alternate;
        };

/* structure for output of get_current_backup_time RPC:
 */
        struct RE_get_current_backup_time_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_time_ty  backupTime;
        };

/* structure for input and output of get_all_backup_times RPC:
 */
        struct RE_get_all_backup_times_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_time_ty  startTime;
                RSTRPC_time_ty  endTime;
                RSTRPC_backup_flags_ty flags;
                long            maxEntries;
                long            cookie;
        };

        struct RE_get_all_backup_times_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_time_list *backupTimes;
                long            numEntries;
                long            cookie;
        };

/* structure for input of is_there_xxxx_backup_for_time and
                                                                set_backup_for_time
 * RPC's:
 */
```

```
        struct RE_backup_for_time_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_time_ty  time;
                RSTRPC_backup_flags_ty flags;
        };

/* structure for input of set_'relative'backup * RPC's: */
        struct RE_set_backup_time_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_backup_flags_ty flags;
        };

/* structure for input and output of get_necessary_media RPC:
 */
        struct RE_get_necessary_media_args {
                RE_rpc_objID    RPCobjID;
                long            maxEntries;
                RSTRPC_bool     all;
                long            cookie;
        };

        struct RE_get_necessary_media_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_media_list *mediaList;
                short           numEntries;
                long            cookie;
        };

/* structures for input and output of is_object_markable RPC:
 */
        struct RE_is_object_markable_args {
                RE_rpc_objID                RPCobjID;
                RSTRPC_user_restorable_object   *thisObject;
        };

        struct RE_is_object_markable_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                bool            markable;
        };

/* structures for input and output of is_object_marked RPC:
 */
        struct RE_is_object_marked_args {
                RE_rpc_objID    RPCobjID;
                RSTRPC_uro_list *objList;
                u_long          numEntries;
        };

        struct RE_is_object_marked_result {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                u_long          numMarked;
                RSTRPC_bool     marked<>;
        };

/* structures for input and output of is_object_searchable and
 * get_backup_times_support RPCs:
 */
        struct RE_tlo_query_args {
                RE_rpc_objID            RPCobjID;
                RSTRPC_top_level_obj    *topLevelObj;
        };

        struct RE_catalog_info {
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                string          level<>;
                string          numrec<>;
                string          catType<>;
        };


/* structure for inputs that require only time
 */
        struct RE_time{
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_time_ty  backupTime;
        };


        struct RE_recx_file_info{
                RE_rpc_objID    RPCobjID;
                RE_errno_ty     status;
                RSTRPC_recx_file_info fileinfo;
        };

program EDM_RESTORE_ENGINE {

   version EDMRE_FUNCTIONS {

            /* rpc for EDMRST_Initialize */
            RE_status_result
                re_initialize( RE_initialize_args ) = 1;

            /* rpc for EDMRST_GetSourceHosts */
            RE_get_hosts_result
                re_get_source_hosts( RE_get_hosts_args ) = 2;

            /* rpc for EDMRST_GetTopLevelObjects */
            RE_get_top_level_objects_result
                re_get_top_level_objects( RE_get_top_level_objects_args ) = 3;

            /* rpc for EDMRST_GetTopLevelTemplates */
            RE_get_top_level_templates_result
                re_get_top_level_templates(
                                    RE_get_top_level_templates_args ) = 4;

            /* rpc for EDMRST_Submit */
            RE_status_result
                re_submit( RE_submit_args ) = 5;

            /* rpc for EDMRST_GetSubmitResults */
            RE_get_submit_results_output
                re_get_submit_results( RE_get_submit_results_args ) = 6;

            /* rpc for EDMRST_Start */
            RE_status_result
                re_start( RE_start_args ) = 7;

            /* rpc for EDMRST_GetRestoreFeedback */
            RE_get_restore_feedback_result
                re_get_restore_feedback( RE_get_restore_feedback_args ) = 8;

            /* rpc for EDMRST_GetQuestion */
            RE_get_question_result
                re_get_question( RE_null_args ) = 9;

            /* rpc for EDMRST_SetUserAnswer */
            RE_status_result
```

```
                re_set_user_answer( RE_set_user_answer_args ) = 10;

            /* rpc for EDMRST_Finish */
            RE_status_result
                re_finish( RE_null_args ) = 11;

            /* rpc for EDMRST_DoesAlternateExist */
           RE_boolean_result
                re_does_alternate_exist( RE_does_alternate_exist_args ) = 12;
            /* rpc's for EDMRST_GetRestorableObjects */
            RE_get_restorable_objects_start_result
                re_get_restorable_objects_start(
                                    RE_get_restorable_objects_start_args ) = 13;
            RE_get_restorable_objects_output_result
                re_get_restorable_objects_output(
                                    RE_get_restorable_objects_output_args ) = 14;

            /* rpc's for EDMRST_FindRestorableObjects */
            RE_find_restorable_objects_result
                re_find_restorable_objects(
                                    RE_find_restorable_objects_args ) = 15;
            RE_get_find_results_result
                re_get_find_results( RE_get_find_results_args ) = 16;

            /* rpc's for EDMRST_MarkObject */
            RE_mark_object_result
                re_mark_object( RE_mark_object_args ) = 17;
            RE_get_mark_results_result
                re_get_mark_results( RE_get_mark_results_args ) = 18;

            /* rpc's for EDMRST_UnmarkObject */
            RE_mark_object_result
                re_unmark_object( RE_unmark_object_args ) = 19;
            RE_get_unmark_results_result
                re_get_unmark_results( RE_get_mark_results_args ) = 20;

            /* rpc for EDMRST_GetMarkedTotalSize */
            RE_get_marked_total_size_result
                re_get_marked_total_size( RE_null_args ) = 21;

            /* rpc for EDMRST_GetCurrentTemplate */
            RE_get_current_template_result
                re_get_current_template( RE_null_args ) = 22;

            /* rpc for EDMRST_GetCurrentBackupTime */
             RE_get_current_backup_time_result
                re_get_current_backup_time( RE_null_args ) = 23;

            /* rpc for EDMRST_GetAllBackupTimes */
             RE_get_all_backup_times_result
                re_get_all_backup_times( RE_get_all_backup_times_args ) = 24;

            /* rpc for EDMRST_IsTherePrevBackup */
             RE_boolean_result
                re_is_there_prev_backup( RE_set_backup_time_args ) = 25;

            /* rpc for EDMRST_IsThereNextBackup */
             RE_boolean_result
                re_is_there_next_backup( RE_set_backup_time_args ) = 26;

            /* rpc for EDMRST_IsTherePrevBackupForTime */
             RE_boolean_result
                re_is_there_prev_backup_for_time(
                                         RE_backup_for_time_args) = 27;

            /* rpc for EDMRST_IsThereNextBackupForTime */
```

```
        RE_boolean_result
           re_is_there_next_backup_for_time(
                                          RE_backup_for_time_args) = 28;

        /* rpc for EDMRST_SetBackupForTime */
         RE_status_result
            re_set_backup_for_time( RE_backup_for_time_args ) = 29;

        /* rpc for EDMRST_SetPrevBackup */
         RE_status_result
            re_set_prev_backup( RE_set_backup_time_args ) = 30;

        /* rpc for EDMRST_SetNextBackup */
         RE_status_result
            re_set_next_backup( RE_set_backup_time_args ) = 31;

        /* rpc for EDMRST_SetFirstBackup */
         RE_status_result
            re_set_first_backup( RE_set_backup_time_args ) = 32;

        /* rpc for EDMRST_SetMostRecentBackup */
         RE_status_result
            re_set_most_recent_backup( RE_set_backup_time_args ) = 33;

        /* rpc for EDMRST_GetNecessaryMedia */
         RE_get_necessary_media_result
            re_get_necessary_media( RE_get_necessary_media_args ) = 34;

        /* rpc for EDMRST_IsObjectMarkable */
         RE_is_object_markable_result
            re_is_object_markable( RE_is_object_markable_args ) = 35;

        /* rpc for EDMRST_IsObjectMarked */
         RE_is_object_marked_result
            re_is_object_marked( RE_is_object_marked_args ) = 36;

        /* rpc for EDMRST_GetDestinationHosts */
        RE_get_hosts_result
            re_get_destination_hosts( RE_get_hosts_args ) = 37;

        /* rpc for EDMRST_GetHostPlatformType */
        RE_get_host_platform_type_result
            re_get_host_platform_type( RE_string_args ) = 38;

        /* rpc for EDMRST_IsObjectSearchable */
        RE_boolean_result
            re_is_object_searchable( RE_tlo_query_args ) = 39;

        /* rpc for EDMRST_GetBackupTimesSupport */
        RE_boolean_result
            re_get_backup_times_support( RE_tlo_query_args ) = 40;

        /* rpc for EDMRE_Load_recx_directives*/
        RE_status_result
            re_load_recx_directives( RE_recx_file_info ) = 41;

        /* rpc for EDMRST_poll_Load_recx_directives */
        RE_status_result
            re_poll_load_recx_directives(RE_null_args ) = 42;

        /* rpc for RSTSL_get_backup_level */
        RE_catalog_info
            re_get_catalog_info(RE_time) = 43;

        /* rpc for EDMRST_GetAllTopLevelObjects */
        RE_get_top_level_objects_result
```

```
            re_get_all_top_level_objects(
                                    RE_get_top_level_objects_args ) = 44;

        /* rpc for EDMRST_GetSymmRestoreOption */
        RE_boolean_result
            re_get_symm_restore_option( RE_tlo_query_args ) = 45;

        /* rpc for EDMRST_Ping */
        RE_status_result
            re_ping( RE_null_args ) = 46;


    } = 1; /* This is version 1 */

%/* This is the RPC program number.
                                  These are reserved in /pds/docs/RPC_numbers
% * This number cannot be re-used by any other RPC daemon on the machine,
                                                                       as it
% * identifies this daemon uniquely.  If it were to be re-used,
                                                             the last daemon
% * to register would be contacted when RPC's come in for this number.
%*/
} = 390016;
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/


/* EDMRestoreEngService.c
 *
 *
 * Mission Statement:  RPC entry points.
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
*/

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: rpcsvc.c,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#define RAW_NETWORK 0
#define PLUGIN 1



#include <esl/c_portable.h>
#include <esl/inout.h>
#include <util/esl_string.h>

#include <logging/logging.h>
#include <csc/cscomm.h>
#include <eerrno/e_eb.h>

#include <restore/csc_EDMRestoreEng.h>
#include <restore/restore_engine.h>
#include <restore/REprogmsg.h>

#include <EDMRestoreEngLog.h>
#include <RSLapi.h>
#include <EDMRECommandApi.h>
#include <restore/EDMREProgressApi.h>
#include <EDMREQuestionApi.h>
#include <EDMRENotifyApi.h>

#include <sys/time.h>

/*
 * External prototypes that are defined locally because of header file
 * conflicts between restore_engine.h and restoreRPC.h
 */

void RSTLL_FreeTimelist( struct RSTRPC_time_list **listhead );
void RSTLL_FreeNameList( struct RSTRPC_name_list **listhead );

/*
 * Local Constants:
 */
/* This constant is designed to allow an asynchronous RPC to complete after
 * an interrupt signal is sent, but not allow the canceling RPC to time out */
#define MAX_CANCEL_WAIT_SECS     20

/* This constant is designed to allow the get_restore_feedback RPC to
 * complete quickly after an interrupt signal is sent, if the cancelation
```

```
 * does not take effect immediately.
 */
#define MAX_CANCEL_RESTORE_WAIT_SECS     1

/*
 * Local functions:
 */
static void set_rpc_obj( ulong rpc_id, RE_rpc_objID *rpc_objID );
static RE_errno_ty check_RPC_state( boolean_ty set, int cmd );
static void clear_RPC_state( void );

/*
 * Local static data:
 */
static int       current_rpc_cmd = COMMAND_NONE_ACTIVE;

/****************************************************************************
**
** Routine:  re_initialize__svc_1
**
** Inputs:   RE_initialize_args * - args for the restore initialize call
**
** Outputs:  None
**
** Return Codes:
**           RE_initialize_result * - result of init function call
**
** Purpose:  Function to create a restore session.
**
** Intended caller:  Internal Only.
****************************************************************************
*/

RE_status_result *
re_initialize_1_svc(IN RE_initialize_args *arg, IN struct svc_req *req )
{
        static RE_status_result argzz;

        setLastRpcTime( );      /* note time of last RPC */
        /* allow multiple calls to Initialize while debugging */
        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else
                argzz.status = RSTSL_Initialize( arg->username);
        if (argzz.status == E_SUCCESS) {
                setGlobalStatus( EDMRE_STATE_BROWSING );
                                                  /* after init is browsing */
                clear_RPC_state( );
        }
        else
                setGlobalStatus( EDMRE_STATE_FAILED );
                                                /* without init, we're dead */

        set_rpc_obj( re_initialize, &argzz.RPCobjID );

        return &argzz;
}

/****************************************************************************
**
** Routine:  re_get_source_hosts
**
** Inputs:   RE_get_hosts_args * - args for the get source hosts call
**
** Outputs:  None
```

```
 **
 ** Return Codes:
 **          RE_get_hosts_result * - result of get source hosts function call
 **
 ** Purpose:  Function to retrieve the backup client hosts
 **
 ** Intended caller:  RPC call from Restore API client
 ***************************************************************************
 */

RE_get_hosts_result *
re_get_source_hosts_1_svc( IN RE_get_hosts_args *arg, IN struct svc_req *req )
{
        static RE_get_hosts_result argzz;
        static RSTRPC_name_list         *hosts = NULL;

        setLastRpcTime( );      /* note time of last RPC */
        if (hosts)
                RSTLL_FreeNameList( &hosts );   /* free old namelist */

        argzz.cookie = arg->cookie;
        argzz.numEntries = 0;
        argzz.hosts = NULL;

        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)          /* if not idle, trouble */
                ;       /* we weren't idle, leave hosts=NULL; reject call */
        else
                argzz.status = RSTSL_GetSourceHosts( arg->hostname,
                                                     arg->maxEntries,
                                                     &hosts,
                                                     &argzz.numEntries,
                                                     &argzz.cookie );

        if (argzz.status == E_SUCCESS)
                argzz.hosts = hosts;

        set_rpc_obj( re_get_source_hosts, &argzz.RPCobjID );

        return &argzz;
}

/***************************************************************************
 **
 ** Routine:  re_get_destination_hosts
 **
 ** Inputs:  RE_get_hosts_args  * - args for the RPC call
 **
 ** Outputs:  None
 **
 ** Return Codes:
 **          RE_get_hosts_result * - result of RPC function call
 **
 ** Purpose:  Function to retrive the names of the possible restore target
                                                                      hosts
 **
 ** Intended caller:  Internal Only.
 ***************************************************************************
 */

RE_get_hosts_result *
re_get_destination_hosts_1_svc(
                         IN RE_get_hosts_args *arg, IN struct svc_req *req )
{
        static RE_get_hosts_result argzz;
        static RSTRPC_name_list         *hosts = NULL;
```

```
        setLastRpcTime( );      /* note time of last RPC */
        if (hosts)
        {
                RSTLL_FreeNameList( &hosts );   /* free old namelist */
        }

        argzz.cookie = arg->cookie;
        argzz.numEntries = 0;
        argzz.hosts = NULL;

        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)          /* if not idle, trouble */
                ;       /* we weren't idle, leave hosts=NULL; reject call */
        else {
                argzz.status = RSTSL_GetDestinationHosts( arg->maxEntries,
                                                          &hosts,
                                                          &argzz.numEntries,
                                                          &argzz.cookie );

                if (E_SUCCESS == argzz.status)
                        argzz.hosts = hosts;
        }

        set_rpc_obj( re_get_destination_hosts, &argzz.RPCobjID );

        return &argzz;
}

/***************************************************************************
 **
 ** Routine:  re_get_top_level_objects
 **
 ** Inputs:   RE_get_top_level_objects_args * - args for the top level objs
                                                                       call
 **
 ** Outputs:  None
 **
 ** Return Codes:
 **          RE_get_top_level_objects_result * - result of function call
 **
 ** Purpose:  Function to retrive the top level objects (
                                                    workitem, workitem sets)
 **
 ** Intended caller:  Internal Only.
 ***************************************************************************
 */

RE_get_top_level_objects_result *
re_get_top_level_objects_1_svc( IN RE_get_top_level_objects_args *arg,
                                IN struct svc_req *req )
{
        static RE_get_top_level_objects_result argzz;
        static short lastNumEntries = 0;
        RSTRPC_tlo_list             *topListPtr;
        RSTRPC_top_level_obj    *tloPtr;

        setLastRpcTime( );      /* note time of last RPC */
        /* free last call's output: */
        if (lastNumEntries) {
                xdr_free( xdr_RE_get_top_level_objects_result, (
                                                             char *)&argzz);
                lastNumEntries = 0;
        }

        argzz.cookie = arg->cookie;
        argzz.numEntries = 0;
        argzz.topLevelObjs = NULL;

        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, leave hosts=NULL; reject call */
        else
                argzz.status = RSTSL_GetTopLevelObjects( arg->sourceHost,
                                                         arg->maxEntries,
                                                         &argzz.topLevelObjs,
                                                         &argzz.numEntries,
                                                         &argzz.cookie,
                                                         PLUGIN);
        lastNumEntries =  argzz.numEntries;

/* Fix returned objects to avoid null string pointers for RPC :  */
        topListPtr = argzz.topLevelObjs;
        while (topListPtr)
        {
                tloPtr = topListPtr->tlo;
                if (!tloPtr->root.objName)
                        tloPtr->root.objName = esl_strdup( "" );
                if (!tloPtr->root.objTypeString)
                        tloPtr->root.objTypeString = esl_strdup( "" );
                if (!tloPtr->fileSpec)
                        tloPtr->fileSpec = esl_strdup( "" );
                if (!tloPtr->templateName)
                        tloPtr->templateName = esl_strdup( "" );
                if (!tloPtr->hostname)
                        tloPtr->hostname = esl_strdup( "" );
                if (!tloPtr->wiBIC)
                        tloPtr->wiBIC = esl_strdup( "" );
#if 0   /* this might cause problem: 0 length, 1 char buffer */
                if (!tloPtr->appData.appData_val)
                        tloPtr->appData.appData_val = esl_strdup( "" );
#endif
                topListPtr = topListPtr->next;
        }

        set_rpc_obj( re_get_top_level_objects, &argzz.RPCobjID );

        return &argzz;
}
/******************************************************************************
**
** Routine:  re_get_all_top_level_objects
**
** Inputs:   RE_get_top_level_objects_args * - args for the top level objs
                                                                        call
**
** Outputs:  None
**
** Return Codes:
**           RE_get_top_level_objects_result * - result of function call
**
** Purpose:  Function to retrive the top level objects (
                                                  workitem, workitem sets)
**
** Intended caller:  Internal Only.
******************************************************************************
*/

RE_get_top_level_objects_result *
re_get_all_top_level_objects_1_svc( IN RE_get_top_level_objects_args *arg,
                                    IN struct svc_req *req )
{
```

```
        static RE_get_top_level_objects_result argzz;
        static short lastNumEntries = 0;
        RSTRPC_tlo_list                 *topListPtr;
        RSTRPC_top_level_obj    *tloPtr;

        setLastRpcTime( );      /* note time of last RPC */
        /* free last call's output: */
        if (lastNumEntries) {
                xdr_free( xdr_RE_get_top_level_objects_result, (
                                                                char *)&argzz);
                lastNumEntries = 0;
        }

        argzz.cookie = arg->cookie;
        argzz.numEntries = 0;
        argzz.topLevelObjs = NULL;

        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, leave hosts=NULL; reject call */
        else
                argzz.status = RSTSL_GetTopLevelObjects( arg->sourceHost,
                                                         arg->maxEntries,
                                                         &argzz.topLevelObjs,
                                                         &argzz.numEntries,
                                                         &argzz.cookie,
                                                         RAW_NETWORK);
        lastNumEntries =  argzz.numEntries;

/* Fix returned objects to avoid null string pointers for RPC :  */
        topListPtr = argzz.topLevelObjs;
        while (topListPtr)
        {
                tloPtr = topListPtr->tlo;
                if (!tloPtr->root.objName)
                        tloPtr->root.objName = esl_strdup( "" );
                if (!tloPtr->root.objTypeString)
                        tloPtr->root.objTypeString = esl_strdup( "" );
                if (!tloPtr->fileSpec)
                        tloPtr->fileSpec = esl_strdup( "" );
                if (!tloPtr->templateName)
                        tloPtr->templateName = esl_strdup( "" );
                if (!tloPtr->hostname)
                        tloPtr->hostname = esl_strdup( "" );
                if (!tloPtr->wiBIC)
                        tloPtr->wiBIC = esl_strdup( "" );
#if 0   /* this might cause problem: 0 length, 1 char buffer */
                if (!tloPtr->appData.appData_val)
                        tloPtr->appData.appData_val = esl_strdup( "" );
#endif
                topListPtr = topListPtr->next;
        }

        set_rpc_obj( re_get_top_level_objects, &argzz.RPCobjID );

        return &argzz;
}

/******************************************************************************
**
** Routine:  re_get_restorable_objects_start
**
** Inputs:   RE_get_restorable_objects_start_args *
**
** Outputs:  None
**
```

```
** Return Codes:
**          RE_get_restorable_objects_start_result *
**
** Purpose:  Function to start the retrieval of the child objects of the
**       specified parent object. The caller specifies the parent object
**       and whether or not to include bad files.
**
** Intended caller:  RPC call from Restore API client
*****************************************************************************
*/

RE_get_restorable_objects_start_result *
re_get_restorable_objects_start_1_svc(
                                 IN RE_get_restorable_objects_start_args *arg,
                                    IN struct svc_req *req )
{
        static RE_get_restorable_objects_start_result   argzz;
        RE_get_restorable_objects_start_args            *cmd_args;

        int     status;

        setLastRpcTime( );      /* note time of last RPC */
        cmd_args = calloc( 1, sizeof(RE_get_restorable_objects_start_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_NO_MEMORY, errno,
                        "Cannot malloc RE_get_restorable_objects_start_args" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no RPC is in progress */
        else if (E_SUCCESS != (argzz.status = check_RPC_state( TRUE,
                                      COMMAND_GET_RESTORABLE_OBJECTS )) )
                ;       /* just return failure status */
        else
        {
                cmd_args->parentObj = arg->parentObj;
                /* change null string template name to NULL ptr */
if (cmd_args->parentObj->objLevel == RSTRPC_tlo_type
 && cmd_args->parentObj->RE_restorable_obj_u.tloInfo->templateName
 && !strlen( cmd_args->parentObj->RE_restorable_obj_u.tloInfo->templateName) )
                {
        free( cmd_args->parentObj->RE_restorable_obj_u.tloInfo->templateName);
        cmd_args->parentObj->RE_restorable_obj_u.tloInfo->templateName = NULL;
                }
                arg->parentObj = NULL;
                cmd_args->cookie =  arg->cookie;
                cmd_args->maxEntries = arg->maxEntries;
                cmd_args->allowBadFiles = arg->allowBadFiles;

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                  "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand(
                                COMMAND_GET_RESTORABLE_OBJECTS, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                  "PushCommand failed");
```

```
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                        argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_get_restorable_objects_start_args,
                                  (char *)cmd_args);
                        free( cmd_args );
                }
        }
        set_rpc_obj( re_get_restorable_objects_start, &argzz.RPCobjID );
        return &argzz;
}


/*****************************************************************************
**
** Routine:  re_get_restorable_objects_output
**
** Inputs:   RE_get_restorable_objects_output_args *
**
** Outputs:  None
**
** Return Codes:
**          RE_get_restorable_objects_output_result *
**
** Purpose:  Function to test for completion of the
**      re_get_restorable_objects_start_1 RPC call, and retrieve some or all
**      of its output.
**
** Intended caller:  RPC call from Restore API client
*****************************************************************************
*/

RE_get_restorable_objects_output_result *
re_get_restorable_objects_output_1_svc(
                                 IN RE_get_restorable_objects_output_args *arg,
                                 IN struct svc_req *req )
{
        static RE_get_restorable_objects_output_result  argzz;
        static RE_get_restorable_objects_output_result  *outarg = NULL;
        int     result, cmd, status;

        setLastRpcTime( );      /* note time of last RPC */
        if (outarg)
        {       /* free last results */
                xdr_free( xdr_RE_get_restorable_objects_output_result,
                          (char *)outarg );
                free( outarg );
                outarg = NULL;
        }

        else
        {       /* init static output struct for errors (
                                                  1st time & aft errs */
                argzz.numEntries = 0;
                argzz.cookie = 0;
                argzz.childrenObjs = NULL;
        }
        /* make sure this RPC is in progress */
        if (E_SUCCESS != (argzz.status = check_RPC_state( FALSE,
                                COMMAND_GET_RESTORABLE_OBJECTS )) )
                ;       /* just return failure status */

        /* test for completion of processing: later use real flag */
        else if (PopResult( -1, &result, &cmd, &status) )
        {
                if (status == COMMAND_RECORD_GET_FAILED)
                {
                        argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                } else {
                        /* log error, clean up, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0, "PopResult failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
        }
        else if (cmd != COMMAND_GET_RESTORABLE_OBJECTS)
        {
                /* log error, clean up, return error  */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_INVALID_COMMAND, 0,
                        "PopResult mismatch: got %d command, expected %d\n",
                                cmd, COMMAND_GET_RESTORABLE_OBJECTS);
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (result != COMMAND_RESULT_SUCCESS)
        {
              EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                        MESSAGE_FAILURE_DOING_ASYNC_RPC , 0,
                        "RPC failure in process manager thread" );
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (PopRpcOutput( (void **)&outarg, &status) )
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status,
                                0, "PopRpcOutput failure");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else
        {       /* return popped results struct */
            set_rpc_obj( re_get_restorable_objects_output, &outarg->RPCobjID);
            clear_RPC_state( );
                                        /* indicate process mgr idle */
            return outarg;
        }

        /* return static result struct on errors */
        set_rpc_obj( re_get_restorable_objects_output, &argzz.RPCobjID );
        if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
            clear_RPC_state( );
                                /* indicate process mgr idle on fatals */
        return &argzz;
}

/*************************************************************************
**
** Routine:  re_mark_object
**
** Inputs:   RE_mark_object_args *
**
** Outputs:  None
**
** Return Codes:
```

```
**          RE_mark_object_result *
**
** Purpose:  Function to start the marking process for a user restorable
**       object and, optionally, for its descendants.
**
** Intended caller:  RPC call from Restore API client
**************************************************************************
*/

RE_mark_object_result *
re_mark_object_1_svc( IN RE_mark_object_args *arg, IN struct svc_req *req )
{
        static RE_mark_object_result    argzz;
        RE_mark_object_args             *cmd_args;

        int     status;

        cmd_args = calloc( 1, sizeof(RE_mark_object_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_NO_MEMORY, errno,
                        "Cannot malloc RE_mark_object_args" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no rpc is in progress */
        else if ( (argzz.status = check_RPC_state(
                                        TRUE, COMMAND_MARK_OBJECT ))
          != E_SUCCESS )
                ;       /* just return failure status */

        else
        {

                ClearRpcCancelFlag( );  /* reset cancel flag */
                ClearProgressValue( );  /* reset progress count */

                cmd_args->thisObj = arg->thisObj;
                arg->thisObj = NULL;    /* so RPC stuff wont free it */
                cmd_args->backupTime =  arg->backupTime;
                cmd_args->allowBadFiles = arg->allowBadFiles;
                cmd_args->descend = arg->descend;

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0,
                                                "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand( COMMAND_MARK_OBJECT, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0,
                                                "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                        argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
```

```
	{	/* failure somewhere: free allocated memeory: */
		if (cmd_args) {
			xdr_free( xdr_RE_mark_object_args, (char *)cmd_args );
			free( cmd_args );
		}
	}

	set_rpc_obj( re_mark_object, &argzz.RPCobjID );
	return &argzz;
}


/*****************************************************************************
**
** Routine:  re_get_mark_results
**
** Inputs:   RE_get_mark_results_args *
**
** Outputs:  None
**
** Return Codes:
**           RE_get_mark_results_result *
**
** Purpose:  Function to test for completion of, or interrupt, the
**       re_mark_object RPC call, and retrieve its output.
**
** Intended caller:  RPC call from Restore API client
*****************************************************************************
*/

RE_get_mark_results_result *
re_get_mark_results_1_svc( IN RE_get_mark_results_args *arg,
                           IN struct svc_req *req )
{
	static RE_get_mark_results_result	argzz;
	static RE_get_mark_results_result	*outarg = NULL;
	int	result, cmd, status;

	setLastRpcTime( );	/* note time of last RPC */
	if (outarg)
	{	/* free last results */
		xdr_free( xdr_RE_get_mark_results_result, (char *)outarg );
		free( outarg );
		outarg = NULL;
	}

	else
	{	/* init static output struct for errors (
							1st time & aft errs */
		argzz.badFileCount = 0;
		argzz.permDenyFileCount = 0;
		argzz.dirMarkCount = 0;
		argzz.fileMarkCount = 0;
		argzz.otherMarkCount = 0;
	}

	/* make sure mark is in progress */
	if ( (argzz.status = check_RPC_state( FALSE, COMMAND_MARK_OBJECT ))
	  != E_SUCCESS )
		;	/* just return failure status */
	/* test for completion of processing: later use real flag */
	else if (PopResult( -1, &result, &cmd, &status) )
	{
		if (status == COMMAND_RECORD_GET_FAILED)
		{
			if (arg->interrupt)
```

```
			{	/* signal cancel, wait till done */
				SetRpcCancelFlag( );
				if (PopResult( MAX_CANCEL_WAIT_SECS, &result,
						&cmd, &status) )
					/* if no result, error */
					argzz.status = EP_RB_RECOVER_SERVERFAIL;
			}
			else {
					argzz.fileMarkCount = ReadProgressValue( );
					argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
			}
		}

		else {
			/* log error, clean up, return error  */
			EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
					status, 0, "PopResult failed");
			argzz.status = EP_RB_RECOVER_SERVERFAIL;
		}
	}

	if (argzz.status != E_SUCCESS)
		;	/* fall thru to error return logic */

	else if (cmd != COMMAND_MARK_OBJECT)
	{
		/* log error, clean up, return error  */
		EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
				MESSAGE_INVALID_COMMAND, 0,
			"PopResult mismatch: got %d command, expected %d\n",
				cmd, COMMAND_MARK_OBJECT);
		argzz.status = EP_RB_RECOVER_SERVERFAIL;
	}
	else if (result != COMMAND_RESULT_SUCCESS)
	{
	       EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
			MESSAGE_FAILURE_DOING_ASYNC_RPC, 0,
			 "RPC failure in process manager thread" );
		argzz.status = EP_RB_RECOVER_SERVERFAIL;
	}
	else if (PopRpcOutput( (void **)&outarg, &status) )
	{
		EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status,
				0, "PopRpcOutput failure");
		argzz.status = EP_RB_RECOVER_SERVERFAIL;
	}
	else
	{	/* return popped results struct */
	    set_rpc_obj( re_get_mark_results, &outarg->RPCobjID);
	    clear_RPC_state( );
						/* indicate process mgr idle */
	    return outarg;
	}

	set_rpc_obj( re_get_mark_results, &argzz.RPCobjID );
	if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
	    clear_RPC_state( );
				/* indicate process mgr idle on fatals */
	return &argzz;
}

/*****************************************************************************
**
** Routine:  re_unmark_object_1
**
** Inputs:   RE_unmark_object_args * - args for the RPC call
```

```
 **
 ** Outputs:  None
 **
 ** Return Codes:
 **           RE_mark_object_result * - result of RPC function call
 **
 ** Purpose:  Function to unmark objects for restoral
 **
 ** Intended caller:  Internal Only.
 ******************************************************************************
 */

RE_mark_object_result *
re_unmark_object_1_svc(IN RE_unmark_object_args *arg, IN struct svc_req *req )
{
        static RE_mark_object_result    argzz;
        RE_unmark_object_args           *cmd_args;
        int                             status;

        setLastRpcTime( );      /* note time of last RPC */
        cmd_args = calloc( 1, sizeof(RE_unmark_object_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_NO_MEMORY, errno,
                        "Cannot malloc RE_unmark_object_args" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no rpc is in progress */
        else if ( (argzz.status = check_RPC_state(
                                        TRUE, COMMAND_UNMARK_OBJECT ))
          != E_SUCCESS )
                ;       /* just return failure status */

        else
        {
                ClearRpcCancelFlag( );  /* reset cancel flag */
                ClearProgressValue( );  /* reset progress count */

                cmd_args->thisObj = arg->thisObj;
                arg->thisObj = NULL;    /* so RPC stuff wont free it */
                cmd_args->backupTime =  arg->backupTime;
                cmd_args->badFilesOnly = arg->badFilesOnly;
                cmd_args->descend = arg->descend;

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0,
                                                "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand( COMMAND_UNMARK_OBJECT, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0,
                                                "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                        argzz.status = E_SUCCESS;
```

```
                }

                if (argzz.status != E_SUCCESS)
                {       /* failure somewhere: free allocated memeory: */
                        if (cmd_args) {
                                xdr_free( xdr_RE_unmark_object_args, (
                                                                char *)cmd_args );
                                free( cmd_args );
                        }
                }


                set_rpc_obj( re_unmark_object, &argzz.RPCobjID );

                return &argzz;
}
/* re_unmark_object_1 */

/******************************************************************************
 **
 ** Routine:  re_get_unmark_results
 **
 ** Inputs:  RE_get_mark_results_args  * - args for the RPC call
 **
 ** Outputs:  None
 **
 ** Return Codes:
 **           RE_get_unmark_results_result * - result of RPC function call
 **
 ** Purpose:  Function to test for completion of the unmark request
 **
 ** Intended caller:  Internal Only.
 ******************************************************************************
 */

RE_get_unmark_results_result *
re_get_unmark_results_1_svc(IN RE_get_mark_results_args *arg,
                                IN struct svc_req *req )
{
        static RE_get_unmark_results_result     argzz;
        static RE_get_unmark_results_result     *outarg = NULL;
        int     result, cmd, status;

        setLastRpcTime( );      /* note time of last RPC */

        if (outarg)
        {       /* free last results */
                xdr_free( xdr_RE_get_unmark_results_result, (char *)outarg );
                free( outarg );
                outarg = NULL;
        }

        else
        {       /* init static output struct for errors (
                                                1st time & aft errs */
                argzz.badFileCount = 0;
                argzz.dirMarkCount = 0;
                argzz.fileMarkCount = 0;
                argzz.otherMarkCount = 0;
        }

        /* make sure unmark is in progress */
        if ( (argzz.status = check_RPC_state(
                                        FALSE, COMMAND_UNMARK_OBJECT ))
          != E_SUCCESS )
                ;       /* just return failure status */
        /* test for completion of processing: later use real flag */
        else if (PopResult( 1, &result, &cmd, &status) )
        {
                if (status == COMMAND_RECORD_GET_FAILED)
                {
                        if (arg->interrupt)
                        {    /* signal cancel, wait till done */
                             SetRpcCancelFlag( );
                             if (PopResult( MAX_CANCEL_WAIT_SECS, &result,
                                         &cmd, &status) )
                                  /* if no result, error */
                                  argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        }
                        else {
                                  argzz.fileMarkCount = ReadProgressValue( );
                                  argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                        }
               }

                else {
                        /* log error, clean up, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                           status, 0, "PopResult failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
        }

        if (argzz.status != E_SUCCESS)
                ;       /* fall thru to error return logic */

        else if (cmd != COMMAND_UNMARK_OBJECT)
        {
                /* log error, clean up, return error  */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                   MESSAGE_INVALID_COMMAND, 0,
                        "PopResult mismatch: got %d command, expected %d\n",
                                   cmd, COMMAND_UNMARK_OBJECT);
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (result != COMMAND_RESULT_SUCCESS)
        {
               EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                   MESSAGE_FAILURE_DOING_ASYNC_RPC, 0,
                           "RPC failure in process manager thread" );
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (PopRpcOutput( (void **)&outarg, &status) )
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status,
                                   0, "PopRpcOutput failure");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else
        {       /* return popped results struct */
            set_rpc_obj( re_get_unmark_results, &outarg->RPCobjID);
            clear_RPC_state( );
                                         /* indicate process mgr idle */
            return outarg;
        }

        set_rpc_obj( re_get_unmark_results, &argzz.RPCobjID );
        if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
            clear_RPC_state( );
                                  /* indicate process mgr idle on fatals */

        return &argzz;
```

```
}
    /* re_get_unmark_results_1 */

/*******************************************************************************
**
** Routine:  re_submit
**
** Inputs:  RE_submit_args   * - args for the RPC call
**
** Outputs: RE_status_result * - result of RPC function call
**
** Purpose:  Function to prepare for the restore of the currently marked
**        objects
**
** Intended caller:  Internal Only.
*******************************************************************************
*/

RE_status_result *
re_submit_1_svc( IN RE_submit_args *arg,
                IN struct svc_req *req )
{
        static RE_status_result         argzz;
        RE_submit_args           *cmd_args;
        int                      status;

        setLastRpcTime( );      /* note time of last RPC */

        cmd_args = calloc( 1, sizeof(RE_submit_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                   MESSAGE_NO_MEMORY, errno,
                      "Cannot malloc RE_submit_args" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no rpc is in progress */
        else if ( (argzz.status = check_RPC_state( TRUE, COMMAND_SUBMIT ))
          != E_SUCCESS )
                ;       /* just return failure status */

        else
        {
                ClearRpcCancelFlag( );  /* reset cancel flag */
                ClearProgressValue( );  /* reset progress count */

                cmd_args->hostname = esl_strdup( arg->hostname );
                cmd_args->directory = esl_strdup( arg->directory );
                cmd_args->overwritePolicy =  arg->overwritePolicy;
                cmd_args->inPlace = arg->inPlace;
                cmd_args->transport = arg->transport;
                cmd_args->submitObjectID = arg->submitObjectID;
                cmd_args->socketClientName = esl_strdup(
                                                  arg->socketClientName);
                cmd_args->socketPort= arg->socketPort;
                cmd_args->mapFile_env = esl_strdup(arg->mapFile_env);
                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                           status, 0,
                                                  "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand( COMMAND_SUBMIT, &status) )
```

```
            {
            /* log error, clean up input queue, return error  */
                    EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                          status, 0,
                                                   "PushCommand failed");
                    PopRpcInput( (void **)&cmd_args, &status);
                    argzz.status = EP_RB_RECOVER_SERVERFAIL;
                    clear_RPC_state( );     /* indicate idle on fatals */
            }
            else
                    argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_submit_args, (char *)cmd_args );
                        free( cmd_args );
                }
        }

    set_rpc_obj( re_submit, &argzz.RPCobjID );

    return &argzz;
}

/*************************************************************************
**
** Routine:  re_get_submit_results
**
** Inputs:  RE_get_submit_results_args  * - args for the RPC call
**
** Outputs: RE_get_submit_results_output * - result of RPC function call
**
** Purpose:  Function to test for completion of the previously started submit
**            operation.
**
** Intended caller:  Internal Only.
*************************************************************************
*/

RE_get_submit_results_output *
re_get_submit_results_1_svc( IN RE_get_submit_results_args *arg,
                             IN struct svc_req *req )
{
        static RE_get_submit_results_output     argzz;
        static RE_get_submit_results_output     *outarg = NULL;
        int     result, cmd, status;

        setLastRpcTime( );      /* note time of last RPC */

        if (outarg)
        {       /* free last results */
                xdr_free( xdr_RE_get_submit_results_output, (char *)outarg );
                free( outarg );
                outarg = NULL;
        }

        else
        {       /* init static output struct for errors (
                                                 1st time & aft errs */
                argzz.submitObjectID = 0;
                argzz.objectsDone = 0;
        }

        /* make sure submit is in progress */
```

```
        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_SUBMIT ))
          != E_SUCCESS )
                ;       /* just return failure status */
        /* test for completion of processing: later use real flag */
        else if (PopResult( -1, &result, &cmd, &status) )
        {
                if (status == COMMAND_RECORD_GET_FAILED)
                {
                        if (arg->interrupt)
                        {    /* signal cancel, wait till done */
                            SetRpcCancelFlag( );
                            if (PopResult( MAX_CANCEL_WAIT_SECS, &result,
                                           &cmd, &status) )
                                /* if no result, error */
                                argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        }
                        else {
                                argzz.objectsDone = ReadProgressValue( );
                                argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                        }
                }

                else {
                        /* log error, clean up, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0, "PopResult failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
        }

        if (argzz.status != E_SUCCESS)
                ;       /* fall thru to error return logic */

        else if (cmd != COMMAND_SUBMIT)
        {
                /* log error, clean up, return error  */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_INVALID_COMMAND, 0,
                        "PopResult mismatch: got %d command, expected %d\n",
                                      cmd, COMMAND_SUBMIT);
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (result != COMMAND_RESULT_SUCCESS)
        {
               EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                     MESSAGE_FAILURE_DOING_ASYNC_RPC, 0,
                               "RPC failure in process manager thread" );
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (PopRpcOutput( (void **)&outarg, &status) )
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status,
                                      0, "PopRpcOutput failure");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else
        {       /* return popped results struct */
            set_rpc_obj( re_get_submit_results, &outarg->RPCobjID);
            clear_RPC_state( );
                                             /* indicate process mgr idle */
            return outarg;
        }

        set_rpc_obj( re_get_submit_results, &argzz.RPCobjID );
        if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
            clear_RPC_state( );
```

```
                                                /* indicate process mgr idle on fatals */

        return &argzz;
}

/*****************************************************************************
**
** Routine:  re_start_1
**
** Inputs:   RE_start_args * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**           RE_status_result * - result of RPC function call
**
** Purpose:  Function to start the restore
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_status_result *
re_start_1_svc(IN RE_start_args *arg, IN struct svc_req *req )
{
        static RE_status_result argzz;
        RE_start_args           *cmd_args;
        int                     status;

        setLastRpcTime( );      /* note time of last RPC */

        cmd_args = calloc( 1, sizeof(RE_start_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_NO_MEMORY, errno,
                                      "Cannot malloc RE_start_args" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no rpc is in progress */
        else if ( (argzz.status = check_RPC_state( TRUE, COMMAND_START ))
          != E_SUCCESS )
                ;       /* just return failure status */

        else
        {
               purgeProgress();
                ClearRpcCancelFlag( );  /* reset cancel flag */
                ClearProgressValue( );  /* reset progress count */

                cmd_args->submitObjectID = arg->submitObjectID;

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                      "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand( COMMAND_START, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
```

```
                                                              "PushCommand failed");
                                PopRpcInput( (void **)&cmd_args, &status);
                                argzz.status = EP_RB_RECOVER_SERVERFAIL;
                                clear_RPC_state( );     /* indicate idle on fatals */
                        }
                        else
                        {
                                setExternalStatus( RE_STATE_STARTING );
                                argzz.status = E_SUCCESS;
                        }
                }

                if (argzz.status != E_SUCCESS)
                {       /* failure somewhere: free allocated memeory: */
                        if (cmd_args) {
                                xdr_free( xdr_RE_start_args, (char *)cmd_args );
                                free( cmd_args );
                        }
                }

                set_rpc_obj( re_start, &argzz.RPCobjID );

                return &argzz;
}

/*****************************************************************************
**
** Routine:  re_get_restore_feedback
**
** Inputs:  RE_get_restore_feedback_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_get_restore_feedback_result * - result of RPC function call
**
** Purpose:  Function to determine the state of an ongoing restore
**        specified time.
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_get_restore_feedback_result *
re_get_restore_feedback_1_svc(IN RE_get_restore_feedback_args *arg,
                              IN struct svc_req *req )
{
        static RE_get_restore_feedback_result   argzz;
        RE_status_result        *outarg = NULL;
        static RE_Notification          *notify = NULL;
        static long             lasttime = 0;
        int     result, cmd, status, ret = 0;
        struct timeval  timeofday;
        void    *dummy = NULL;

        setLastRpcTime( );      /* note time of last RPC */

        /* init static output struct for progress */
        if (NULL != notify)     /* release old feedback */
                xdr_free( xdr_RE_get_restore_feedback_result, (
                                                                  char *)&argzz );
        memset( &argzz, 0, sizeof(RE_get_restore_feedback_result) );
        if (NULL == (notify = calloc( 1, sizeof(RE_Notification) )))
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_NO_MEMORY, errno,
                              "Cannot malloc RE_Notification" );
            argzz.status = EP_RB_RECOVER_NOMEM;

            set_rpc_obj( re_get_restore_feedback, &argzz.RPCobjID );

            return &argzz;
        }

        /* make sure restore (start) is in progress */
        if ( (argzz.status = check_RPC_state(
                                  FALSE, COMMAND_START )) == E_SUCCESS )
        {
            /* test for completion of processing: later use real flag */
            if ( (ret = PopResult( -1, &result, &cmd, &status)) != 0 )
            {
                if (status == COMMAND_RECORD_GET_FAILED)
                {   /* set cancel if requested */
                    if (arg->quit_restore)
                    {
                        SetRpcCancelFlag( );
                        if ( (ret = PopResult( MAX_CANCEL_RESTORE_WAIT_SECS,
                                              &result, &cmd,
                                                      &status)) != 0 )
                        {
                            /* if no result, user must keep trying */
                           argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                        }
                        else { /* result popped, leave E_SUCCESS to */
                               /* update (final) stats below */
                        }
                    }
                    else        /* no cancel and not done already */
                    {
                        argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                    }
                }
                else {
                    /* log error, clean up, return error  */
                    EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                         status, 0, "PopResult failed");
                    argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
            }

            if (ret == 0)
           {
                if (cmd != COMMAND_START)
                {
                    /* log error, clean up, return error  */
                    EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                         MESSAGE_INVALID_COMMAND, 0,
                          "PopResult mismatch: got %d command,
                                                      expected %d\n",
                                         cmd, COMMAND_START);
                    argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
                else if (result != COMMAND_RESULT_SUCCESS)
                {
                   EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        MESSAGE_FAILURE_DOING_ASYNC_RPC,
                                                                      0,
                               "RPC failure in process manager thread"
                                                                      );
                    argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
```

```
                if (PopRpcOutput( (void **)&outarg, &status) )
                {
                        EDMRestoreEng_logent(
                                     __FILE__, __LINE__, LOG_ERR, status,
                                            0, "PopRpcOutput failure");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
               else
               {
                        argzz.status = outarg->status;
                        xdr_free( xdr_RE_status_result, (char *)outarg );
                        free( outarg );
                }

                clear_RPC_state( );
                                          /* indicate process mgr idle */
                lasttime = 0;   /* in case multiple starts possible later */
                setGlobalStatus (EDMRE_STATE_BROWSING);
                                                     /* back to browsing */
            }

            if (argzz.status == EP_RB_RECOVER_SERVERFAIL) {
                clear_RPC_state( );
                                 /* indicate process mgr idle on fatals */
                lasttime = 0;   /* in case multiple starts possible later */
                setGlobalStatus (EDMRE_STATE_BROWSING);
                                                     /* back to browsing */
            }
        }

        gettimeofday( &timeofday, dummy );
                                      /* for time of getRestoreStatus */

        if (0 != getRestoreStatus( lasttime, &argzz.rstStats, &status ))
        {
                /* log error, continue */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status, 0,
                                     "getRestoreStatus failed");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }

       if (argzz.status == EP_RB_RECOVER_RPC_INCOMPLETE)
        {
                lasttime = timeofday.tv_sec - 120;
        }

        ret = PopNotifications( notify, &status );

       if (ret == 0)
       {
            argzz.notify = notify;
        }

        set_rpc_obj( re_get_restore_feedback, &argzz.RPCobjID );

        return &argzz;
}
        /* end of re_get_restore_feedback_1 */

/****************************************************************************
**
** Routine:  re_get_question
**
** Inputs:  RE_null_args  * - args for the RPC call (none)
**
** Outputs:  None
```

```
 **
 ** Return Codes:
 **          RE_get_question_result * - result of RPC function call
 **
 ** Purpose:  Function to retrieve a restore execution query
 **
 ** Intended caller:  Internal Only.
 *******************************************************************************
 */

RE_get_question_result *
re_get_question_1_svc( IN RE_null_args *arg, IN struct svc_req *req )
{
        static RE_get_question_result argzz;
        static Question                 question;
        int     result, status;

        setLastRpcTime( );      /* note time of last RPC */

        argzz.query = NULL;     /* init response structure */
        /* dont free last question - its owned by process thread.
                                                           This is copy*/
        memset( &question, 0, sizeof(Question) );

        /* make sure restore (start) is in progress */
        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_START ))
          != E_SUCCESS )
                ;       /* just return failure status */

        else if (getExternalStatus() != RE_STATE_STOPPED)
        {       /* not awaiting answer, either user error or aborted */
                argzz.status = EP_RB_RECOVER_INVALOP;
        }

        /* in proper state: fetch question from question queue */

        else if ( 0 != (result = PopQuestion( 1, &question, &status ) ) )
        {       /* dequeue question failed -- log error, continue */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status, 0,
                                        "PopQuestion failed");
                if (
                status == QUESTION_RECORD_GET_FAILED) /* assume user wrong */
                        argzz.status = EP_RB_RECOVER_INVALOP;
                else    /* internal error */
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else
                argzz.query = &question;
                                        /* return question structure */

        set_rpc_obj( re_get_question, &argzz.RPCobjID );

        return &argzz;
}

/*******************************************************************************
**
** Routine:  re_set_user_answer
**
** Inputs:  RE_set_user_answer_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
```

```
** Purpose:  Function to return the user response to a restore execution query
**
** Intended caller:  Internal Only.
*******************************************************************************
*/

RE_status_result *
re_set_user_answer_1_svc(IN RE_set_user_answer_args *arg,
                         IN struct svc_req *req )
{
        static RE_status_result argzz;
        int     status;

        setLastRpcTime( );      /* note time of last RPC */

        /* make sure restore (start) is in progress */
        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_START ))
          != E_SUCCESS )
                ;       /* just return failure status */

        else if (getExternalStatus() != RE_STATE_STOPPED)
        {       /* not awaiting answer, either user error or aborted */
                argzz.status = EP_RB_RECOVER_INVALOP;
        }

        /* in proper state: push response on answer queue */

        else if ( PushAnswer( &arg->answers, &status ) )
        {       /* enqueue failed -- log error, continue */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status, 0,
                                        "PushAnswer failed");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }

        else
        {       /* restore external state to proper phase */
                if ( EDMRE_STATE_PREPHASE == getGlobalStatus(NULL) )
                        setExternalStatus(RE_STATE_PREPHASE);
                else
                        setExternalStatus(RE_STATE_POSTPHASE);
                /* clear answer list pointer,
                                        since its now on answer queue */
                arg->answers.firstanswer = NULL;
                                                /* so only freed once */
        }

        set_rpc_obj( re_set_user_answer, &argzz.RPCobjID );

        return &argzz;
}

/*******************************************************************************
**
** Routine:  re_get_top_level_templates_1
**
** Inputs:   RE_get_top_level_templates_args * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_get_top_level_templates_result * - result of RPC function call
**
** Purpose:  Function to retrieve templates configured for the current top
                                                                        level
**           backup object.
```

```
 **
 ** Intended caller:  Internal Only.
 ****************************************************************************
 */

RE_get_top_level_templates_result *
re_get_top_level_templates_1_svc( IN RE_get_top_level_templates_args *arg,
                                  IN struct svc_req *req )
{
        static RE_get_top_level_templates_result argzz;
        static short    lastNumEntries = 0;

        setLastRpcTime( );      /* note time of last RPC */

        /* free last call's output: */
        if (lastNumEntries) {
                xdr_free( xdr_RE_get_top_level_templates_result, (
                                                        char *)&argzz);
                lastNumEntries = 0;
        }

        argzz.cookie = arg->cookie;
        argzz.numEntries = 0;
        argzz.templates = NULL;

        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, leave templates=NULL;
                                                        reject call */
        else {
                argzz.status = RSTSL_GetTopLevelTemplates( arg->topLevelObj,
                                                arg->maxEntries,
                                                &argzz.templates,
                                                &argzz.numEntries,
                                                &argzz.cookie );
                lastNumEntries = argzz.numEntries;
        }

        set_rpc_obj( re_get_top_level_templates, &argzz.RPCobjID );

        return &argzz;
}

/****************************************************************************
 **
 ** Routine:  re_get_current_template
 **
 ** Inputs:  RE_null_args  * - args for the RPC call (none)
 **
 ** Outputs:  None
 **
 ** Return Codes:
 **          RE_get_current_template_result * - result of RPC function call
 **
 ** Purpose:  Function to retrieve trhe currently selected template name
 **
 ** Intended caller:  Internal Only.
 ****************************************************************************
 */

RE_get_current_template_result *
re_get_current_template_1_svc(IN RE_null_args *arg,
                              IN struct svc_req *req )
{
        static RE_get_current_template_result argzz;
        static char     template_buff[MAX_TEMPLATE_LEN] = "";
```

```
        setLastRpcTime( );      /* note time of last RPC */

        /* init output struct ptr first time; clear string other times */
        if (template_buff[0] == 0)
                argzz.templateName = template_buff;
        else
                template_buff[0] = 0;

        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_GetCurrentTemplate( argzz.templateName,
                                                        &argzz.alternate );
        }

        set_rpc_obj( re_get_current_template, &argzz.RPCobjID );

        return &argzz;
}

/****************************************************************************
 **
 ** Routine:  re_get_necessary_media
 **
 ** Inputs:  RE_get_necessary_media_args  * - args for the RPC call
 **
 ** Outputs:  None
 **
 ** Return Codes:
 **          RE_get_necessary_media_result * - result of RPC function call
 **
 ** Purpose:  Function to retrieve the list of media need to restore the
 **       currently marked objects
 **
 ** Intended caller:  Internal Only.
 ****************************************************************************
 */

RE_get_necessary_media_result *
re_get_necessary_media_1_svc(IN RE_get_necessary_media_args *arg,
                             IN struct svc_req *req )
{
        static RE_get_necessary_media_result argzz;
        static RSTRPC_media_list        *media_list = NULL;

        setLastRpcTime( );      /* note time of last RPC */

        /* free previosly returned list of media */
        if (media_list) {
                RSTSL_FreeMediaObjectList( media_list );
                media_list = NULL;
        }

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        else if ( (argzz.status = check_RPC_state(
                                        FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else {
                /* init result structure */
                argzz.numEntries = 0;
                argzz.cookie = arg->cookie;
                argzz.mediaList = NULL;

                argzz.status = RSTSL_GetNecessaryMedia( arg->maxEntries,
                                                        &argzz.mediaList,
                                                        &argzz.numEntries,
                                                        arg->all,
                                                        &argzz.cookie );

                media_list = argzz.mediaList;
                                                /* save to free next time in */
        }

    set_rpc_obj( re_get_necessary_media, &argzz.RPCobjID );

    return &argzz;
}

/*****************************************************************************
**
** Routine:  re_get_all_backup_times
**
** Inputs:  RE_get_all_backup_times_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to start the asynchronous operation to find all the
**          backups available for the current workitem
**
** Intended caller:  RPC call from Restore API client
*****************************************************************************
*/

RE_status_result *
re_get_all_backup_times_1_svc(IN RE_get_all_backup_times_args *arg,
                              IN struct svc_req *req )
{
        static RE_status_result              argzz;
        RE_get_all_backup_times_args    *cmd_args;
        int                             status;
        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        cmd_args = calloc( 1, sizeof(RE_get_all_backup_times_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_NO_MEMORY, errno,
                                "Cannot malloc RE_get_all_backup_times_args"
                                                                        );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no RPC is in progress */
        else if (E_SUCCESS != (argzz.status = check_RPC_state( TRUE,
                                      COMMAND_GET_ALL_TIMES )) )
                ;       /* just return failure status */
        else {
                cmd_args->startTime = arg->startTime;
                cmd_args->endTime = arg->endTime;
                cmd_args->flags = arg->flags;
                cmd_args->maxEntries = arg->maxEntries;
```

```
                cmd_args->cookie = arg->cookie;

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                  "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand( COMMAND_GET_ALL_TIMES, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                  "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                        argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_get_all_backup_times_args,
                                  (char *)cmd_args);
                        free( cmd_args );
                }
        }

        set_rpc_obj( re_get_all_backup_times, &argzz.RPCobjID );

        return &argzz;
}

/*****************************************************************************
**
** Routine:  re_get_all_backup_times_result
**
** Inputs:  RE_null_args  * - No args for this RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_get_all_backup_times_result * - result of RPC function call
**
** Purpose:  Function to test for completion of the re_get_all_backup_times
**           RPC call, and retrieve some or all of its output.
**
** Intended caller:  RPC call from Restore API client
*****************************************************************************
*/

RE_get_all_backup_times_result *
re_get_all_backup_times_result_1_svc( IN RE_null_args *arg,
                                      IN struct svc_req *req )
{

        static RE_get_all_backup_times_result argzz;
        static RE_get_all_backup_times_result *outarg = NULL;
        int     result, cmd, status;
        setLastRpcTime( );      /* note time of last RPC */

        if (outarg)
        {       /* free last results */
                outarg->backupTimes = NULL;  /* this is freed by RSTSL_... */
                xdr_free( xdr_RE_get_all_backup_times_result,
                        (char *)outarg );
                free( outarg );
                outarg = NULL;
        }

        else
        {       /* init static output struct for errors (
                                                1st time & aft errs */
                argzz.numEntries = 0;
                argzz.cookie = 0;
                argzz.backupTimes = NULL;
        }

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        /* make sure this RPC is in progress */
        if (E_SUCCESS != (argzz.status = check_RPC_state( FALSE,
                                        COMMAND_GET_ALL_TIMES )) )
                ;       /* just return failure status */

        /* test for completion of processing */
        else if (PopResult( -1, &result, &cmd, &status) )
        {
                if (status == COMMAND_RECORD_GET_FAILED)
                {
                        argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                } else {
                        /* log error, clean up, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0, "PopResult failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
        }
        else if (cmd != COMMAND_GET_ALL_TIMES)
        {
                /* log error, clean up, return error  */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_INVALID_COMMAND, 0,
                        "PopResult mismatch: got %d command, expected %d\n",
                                cmd, COMMAND_GET_ALL_TIMES);
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (result != COMMAND_RESULT_SUCCESS)
        {
              EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                          MESSAGE_FAILURE_DOING_ASYNC_RPC , 0,
                          "RPC failure in process manager thread" );
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (PopRpcOutput( (void **)&outarg, &status) )
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status,
                                0, "PopRpcOutput failure");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else
        {       /* return popped results struct */
            set_rpc_obj( re_get_all_backup_times_result, &outarg->RPCobjID);
            clear_RPC_state( );
```

```
                                                /* indicate process mgr idle */
                return outarg;
        }

        /* return static result struct on errors */
        set_rpc_obj( re_get_all_backup_times_result, &argzz.RPCobjID );
        if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
            clear_RPC_state( );
                                        /* indicate process mgr idle on fatals */
        return &argzz;
}

/*****************************************************************************
**
** Routine:  re_get_current_backup_time
**
** Inputs:  RE_null_args   * - args for the RPC call (none)
**
** Outputs:  None
**
** Return Codes:
**          RE_get_current_backup_time_result * - result of RPC function call
**
** Purpose:  Function to retrieve the currently selected backup time
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_get_current_backup_time_result *
re_get_current_backup_time_1_svc(
                                IN RE_null_args *arg, IN struct svc_req *req )
{
        static RE_get_current_backup_time_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        /* init result structure */
        argzz.backupTime = 0;

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        else if ( (argzz.status = check_RPC_state(
                                        FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_GetCurrentBackupTime(
                                                &argzz.backupTime );
        }

        set_rpc_obj( re_get_current_backup_time, &argzz.RPCobjID );

        return &argzz;
}

/*****************************************************************************
**
** Routine:  re_is_there_prev_backup
**
** Inputs:  RE_set_backup_time_args   * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
```

```
**          RE_boolean_result * - result of RPC function call
**
** Purpose:  Function to test if there is an older backup available
**
** Intended caller:  Internal Only.
*******************************************************************************
*/

RE_boolean_result *
re_is_there_prev_backup_1_svc( IN RE_set_backup_time_args *arg,
                               IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        else if ( (argzz.status = check_RPC_state(
                                         FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_IsTherePrevBackup( arg->flags,
                                                 &argzz.boolResult );
        }

        set_rpc_obj( re_is_there_prev_backup, &argzz.RPCobjID );

        return &argzz;
}

/******************************************************************************
**
** Routine:  re_is_there_next_backup_for_time
**
** Inputs:  RE_backup_for_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_boolean_result * - result of RPC function call
**
** Purpose:  Function to test if there is a backup available more recent than
                                                                              a
**       specified time.
**
** Intended caller:  Internal Only.
*******************************************************************************
*/

RE_boolean_result *
re_is_there_next_backup_for_time_1_svc(IN RE_backup_for_time_args *arg,
                                IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        else if ( (argzz.status = check_RPC_state(
                                         FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
```

```
                ;       /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_IsThereNextBackupForTime( arg->time,
                                                        arg->flags,
                                                       &argzz.
                                                         boolResult );
        }

        set_rpc_obj( re_is_there_next_backup_for_time, &argzz.RPCobjID );

        return &argzz;
}

/******************************************************************************
**
** Routine:  re_is_there_next_backup
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_boolean_result * - result of RPC function call
**
** Purpose:  Function to test if there is a backup available more recent than
**       the currently selected time.
**
** Intended caller:  Internal Only.
*******************************************************************************
*/

RE_boolean_result *
re_is_there_next_backup_1_svc(IN RE_set_backup_time_args *arg,
                              IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        else if ( (argzz.status = check_RPC_state(
                                         FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_IsThereNextBackup( arg->flags,
                                                 &argzz.boolResult );
        }

        set_rpc_obj( re_is_there_next_backup, &argzz.RPCobjID );

        return &argzz;
}

/******************************************************************************
**
** Routine:  set_backup_time_request
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**          int         internal_command
**          int rpc_function_no
**
** Outputs:  None
**
```

```
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to start the asynchronous operation of all the
**       re_set..._backup rpc functions
**
** Intended caller:  RPC function service calls
***********************************************************************
*/

RE_status_result *
set_backup_time_request(IN RE_set_backup_time_args *arg,
                        IN int internal_command,
                        IN int rpc_function_no )
{
        static RE_status_result                argzz;
        RE_set_backup_time_args                *cmd_args;
        int                            status;
        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        cmd_args = calloc( 1, sizeof(RE_set_backup_time_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                   MESSAGE_NO_MEMORY, errno,
                            "Cannot malloc RE_set_backup_time_args" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no RPC is in progress */
        else if (E_SUCCESS != (argzz.status = check_RPC_state( TRUE,
                                 internal_command )) )
                ;       /* just return failure status */
        else {
                cmd_args->flags = arg->flags;

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                           status, 0,
                                                   "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand( internal_command, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                           status, 0,
                                                   "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                        argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_set_backup_time_args,
                                (char *)cmd_args);
```

```
                        free( cmd_args );
                }
        }

        set_rpc_obj( rpc_function_no, &argzz.RPCobjID );

        return &argzz;
}

/***********************************************************************
**
** Routine:  set_backup_time_result
**
** Inputs:  int  internal_command
**          int rpc_function_no
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to test for completion of the re_set_xxx_backup
**           RPC calls, and retrieve some or all of their output.
**
** Intended caller:  RPC service function
***********************************************************************
*/

RE_status_result *
set_backup_time_result( IN int internal_command, IN int rpc_function_no )
{

        static RE_status_result argzz;
        static RE_status_result *outarg = NULL;
        int     result, cmd, status;

        setLastRpcTime( );      /* note time of last RPC */

        if (outarg)
        {       /* free last results */
                xdr_free( xdr_RE_status_result, (char *)outarg );
                free( outarg );
                outarg = NULL;
        }

        /* make sure this RPC is in progress */
        if (E_SUCCESS != (argzz.status = check_RPC_state( FALSE,
                                                  internal_command )) )
                ;       /* just return failure status */

        /* test for completion of processing */
        else if (PopResult( -1, &result, &cmd, &status) )
        {
                if (status == COMMAND_RECORD_GET_FAILED)
                {
                        argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                } else {
                        /* log error, clean up, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                           status, 0, "PopResult failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
        }
        else if (cmd != internal_command)
        {
                /* log error, clean up, return error  */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_INVALID_COMMAND, 0,
                            "PopResult mismatch: got %d command, expected %d\n",
                                      cmd, internal_command);
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (result != COMMAND_RESULT_SUCCESS)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                  MESSAGE_FAILURE_DOING_ASYNC_RPC , 0,
                                  "RPC failure in process manager thread" );
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (PopRpcOutput( (void **)&outarg, &status) )
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status,
                                      0, "PopRpcOutput failure");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else
        {       /* return popped results struct */
            set_rpc_obj( rpc_function_no, &outarg->RPCobjID);
            clear_RPC_state( );
                                                /* indicate process mgr idle */
            return outarg;
        }

        /* return static result struct on errors */
        set_rpc_obj( rpc_function_no, &argzz.RPCobjID );
        if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
            clear_RPC_state( );
                                        /* indicate process mgr idle on fatals */
        return &argzz;
}

/**************************************************************************
**
** Routine:  re_set_first_backup
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**           RE_status_result * - result of RPC function call
**
** Purpose:  Function to select the oldest backup for the current workitem
**
** Intended caller:  Internal Only.
***************************************************************************
*/

RE_status_result *
re_set_first_backup_1_svc(
                     IN RE_set_backup_time_args *arg, IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_request( arg,
                                         COMMAND_SET_FIRST_BACKUP,
                                         re_set_first_backup );

        return argzz;
}

/**************************************************************************
```

```
**
** Routine:  re_set_first_backup_result
**
** Inputs:  RE_null_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**           RE_status_result * - result of RPC function call
**
** Purpose:  Function to test for completion of the rpc to select the oldest
**       backup for the current workitem
**
** Intended caller:  Internal Only.
***************************************************************************
*/

RE_status_result *
re_set_first_backup_result_1_svc( IN RE_null_args *arg,
                                  IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_result( COMMAND_SET_FIRST_BACKUP,
                                        re_set_first_backup );

        return argzz;
}


/**************************************************************************
**
** Routine:  re_set_next_backup
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**           RE_status_result * - result of RPC function call
**
** Purpose:  Function to set to the next (more recent) backup time
**
** Intended caller:  Internal Only.
***************************************************************************
*/

RE_status_result *
re_set_next_backup_1_svc(
                     IN RE_set_backup_time_args *arg, IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_request( arg,
                                         COMMAND_SET_NEXT_BACKUP,
                                         re_set_next_backup );

        return argzz;
}
/**************************************************************************
**
** Routine:  re_set_next_backup_result
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
```

```
 **
 ** Return Codes:
 **          RE_status_result * - result of RPC function call
 **
 ** Purpose:  Function to set to the next (more recent) backup time
 **
 ** Intended caller:  Internal Only.
 ************************************************************************
 */

RE_status_result *
re_set_next_backup_result_1_svc(
                                IN RE_null_args *arg, IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_result( COMMAND_SET_NEXT_BACKUP,
                                        re_set_next_backup );

        return argzz;
}


/************************************************************************
**
** Routine:  re_set_prev_backup
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to set to the next (more recent) backup time
**
** Intended caller:  Internal Only.
************************************************************************
*/

RE_status_result *
re_set_prev_backup_1_svc(
                     IN RE_set_backup_time_args *arg, IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_request( arg,
                                         COMMAND_SET_PREVIOUS_BACKUP,
                                         re_set_prev_backup );

        return argzz;
}
/************************************************************************
**
** Routine:  re_set_prev_backup_result
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to set to the next (more recent) backup time
**
** Intended caller:  Internal Only.
```

```
*************************************************************************
*/

RE_status_result *
re_set_previous_backup_result_1_svc(
                                IN RE_null_args *arg, IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_result( COMMAND_SET_PREVIOUS_BACKUP,
                                        re_set_prev_backup );

        return argzz;
}

/************************************************************************
**
** Routine:  re_set_backup_for_time
**
** Inputs:  RE_backup_for_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to set to a specified backup time.
**
** Intended caller:  Internal Only.
************************************************************************
*/

RE_status_result *
re_set_backup_for_time_1_svc( IN RE_backup_for_time_args *arg,
                              IN struct svc_req *req )
{



        static RE_status_result                 argzz;
        RE_backup_for_time_args                 *cmd_args;
        int                                 status;
        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        cmd_args = calloc( 1, sizeof(RE_backup_for_time_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_NO_MEMORY, errno,
                               "Cannot malloc RE_get_all_backup_times_args"
                                                                          );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no RPC is in progress */
        else if (E_SUCCESS != (argzz.status = check_RPC_state( TRUE,
                                        COMMAND_SET_BACKUP_FOR_TIME )) )
                ;       /* just return failure status */
        else {
                cmd_args->flags = arg->flags;
                cmd_args->time = arg->time;

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                                status, 0,
                                                        "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand( COMMAND_SET_BACKUP_FOR_TIME, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                                status, 0,
                                                        "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                        argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_backup_for_time_args,
                                (char *)cmd_args);
                        free( cmd_args );
                }
        }

        set_rpc_obj( re_set_backup_for_time, &argzz.RPCobjID );

        return &argzz;
}
/*****************************************************************************
**
** Routine:  re_set_backup_for_time_result
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to set to the next (more recent) backup time
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_status_result *
re_set_backup_for_time_result_1_svc(
                        IN RE_null_args *arg, IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_result( COMMAND_SET_BACKUP_FOR_TIME,
                                        re_set_backup_for_time );

        return argzz;
}

/*****************************************************************************
**
```

```
** Routine:  re_is_there_prev_backup_for_time_1
**
** Inputs:   RE_backup_for_time_args * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_boolean_result * - result of RPC function call
**
** Purpose:  Function to determine if there is an older backup available.
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_boolean_result *
re_is_there_prev_backup_for_time_1_svc(IN RE_backup_for_time_args *arg,
                                        IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        else if ( (argzz.status = check_RPC_state(
                                        FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_IsTherePrevBackupForTime( arg->time,
                                                                arg->flags,
                                                              &argzz.
                                                                boolResult );

        }

        set_rpc_obj( re_is_there_prev_backup_for_time, &argzz.RPCobjID );

        return &argzz;
}



/*****************************************************************************
**
** Routine:  re_set_most_recent_backup
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to set to the next (more recent) backup time
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_status_result *
re_set_most_recent_backup_1_svc(
                        IN RE_set_backup_time_args *arg, IN struct svc_req *req )
```

```
{
        RE_status_result *argzz;

        argzz = set_backup_time_request( arg,
                                         COMMAND_SET_MOST_RECENT_BACKUP,
                                          re_set_prev_backup );

        return argzz;
}
/**************************************************************************
**
** Routine:  re_set_most_recent_backup_result
**
** Inputs:  RE_set_backup_time_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to set to the next (more recent) backup time
**
** Intended caller:  Internal Only.
**************************************************************************
*/

RE_status_result *
re_set_most_recent_backup_result_1_svc(
                                IN RE_null_args *arg, IN struct svc_req *req )
{
        RE_status_result *argzz;

        argzz = set_backup_time_result( COMMAND_SET_MOST_RECENT_BACKUP,
                                        re_set_most_recent_backup );

        return argzz;
}

/**************************************************************************
**
** Routine:  re_get_host_platform_type_1
**
** Inputs:   RE_string_args * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_get_host_platform_type_result * - result of RPC function call
**
** Purpose:  Function to retrieve the platform type of the specified host
**
** Intended caller:  Internal Only.
**************************************************************************
*/

RE_get_host_platform_type_result *
re_get_host_platform_type_1_svc(
                                IN RE_string_args *arg, IN struct svc_req *req )
{
        static RE_get_host_platform_type_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;
```

```
        else if ( (argzz.status = check_RPC_state(
                                          FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
              ;         /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_GetHostPlatformType( arg->name,
                                                          &argzz.ptype );
        }

        set_rpc_obj( re_get_host_platform_type, &argzz.RPCobjID );

        return &argzz;
}

/**************************************************************************
**
** Routine:  re_does_alternate_exist
**
** Inputs:  RE_does_alternate_exist_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_boolean_result * - result of RPC function call
**
** Purpose:  Function to test if there is an alternate backup trailset
**       available for the specified template
**
** Intended caller:  Internal Only.
**************************************************************************
*/

RE_boolean_result *
re_does_alternate_exist_1_svc( IN RE_does_alternate_exist_args *arg,
                               IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        if (NULL == arg)
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

        else if ( (argzz.status = check_RPC_state(
                                          FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
              ;         /* we weren't idle, reject call */
        else {
                argzz.status = RSTSL_DoesAlternateExist( arg->topLevelObj,
                                                         arg->templateName,
                                                         &argzz.boolResult );
        }

        set_rpc_obj( re_does_alternate_exist, &argzz.RPCobjID );

        return &argzz;
}

/**************************************************************************
**
** Routine:  re_finish_1
**
** Inputs:   RE_null_args * - args for the RPC call (none)
**
** Outputs:  None
**
```

```
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to terminate the restore session at the browse stage
**
** Intended caller:  Internal Only.
***************************************************************************
*/

RE_status_result *
re_finish_1_svc(IN RE_null_args *arg, IN struct svc_req *req )
{
        static RE_status_result argzz;
        RE_null_args    *cmd_args;
        int     status;

        setLastRpcTime( );      /* note time of last RPC */

        cmd_args = calloc( 1, sizeof(RE_null_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_NO_MEMORY, errno,
                                      "Cannot malloc RE_null_args" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        else if ( (argzz.status = check_RPC_state(
                                        TRUE, COMMAND_NONE_ACTIVE ) )
                != E_SUCCESS)   /* if idle, stay idle */
                ;       /* we weren't idle, reject finish */
        else
        {
                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                      "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
                else if (PushCommand( COMMAND_FINISH, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                      "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
                else
                        argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_null_args, (char *)cmd_args );
                        free( cmd_args );
                }
        }

        set_rpc_obj( re_finish, &argzz.RPCobjID );

        return &argzz;
}
```

```
/***************************************************************************
**
** Routine:  re_ping_1
**
** Inputs:   RE_null_args * - args for the RPC call (none)
**
** Outputs:  None
**
** Return Codes:
**          RE_status_result * - result of RPC function call
**
** Purpose:  Function to keep the engine alive
**
** Intended caller:  Internal Only.
***************************************************************************
*/

RE_status_result *
re_ping_1_svc(IN RE_null_args *arg, IN struct svc_req *req )
{
  static RE_status_result argzz;

  setLastRpcTime( );     /* note time of last RPC */

  argzz.status = E_SUCCESS;

  return &argzz;
}


/***************************************************************************
**
** Routine:  re_get_marked_total_size
**
** Inputs:  RE_null_args  * - args for the RPC call (none)
**
** Outputs:  None
**
** Return Codes:
**          RE_get_marked_total_size_result * - result of RPC function call
**
** Purpose:  Function to return the total size of the objects currently marked
**        for restore
**
** Intended caller:  Internal Only.
***************************************************************************
*/

RE_get_marked_total_size_result *
re_get_marked_total_size_1_svc(IN RE_null_args *arg, IN struct svc_req *req )
{
        static RE_get_marked_total_size_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        argzz.total.high = 0;
        argzz.total.low = 0;

        if ( (argzz.status = check_RPC_state( FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
              ;       /* we weren't idle, reject call */
        else
        {
                argzz.total = RSTSL_GetMarkedTotalSize( );
                argzz.status = E_SUCCESS;
        }

    set_rpc_obj( re_get_marked_total_size, &argzz.RPCobjID );

    return &argzz;
}

/*****************************************************************************
**
** Routine:  re_is_object_marked_1
**
** Inputs:   RE_is_object_marked_args * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_is_object_marked_result * - result of RPC function call
**
** Purpose:  Function to determine if specified object is marked for restore
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_is_object_marked_result *
re_is_object_marked_1_svc(
                IN RE_is_object_marked_args *arg, IN struct svc_req *req )
{
        static RE_is_object_marked_result argzz;
        static marked_len = 0;

        setLastRpcTime( );      /* note time of last RPC */

        /* free previosly calloc's bool array */
        if (marked_len) {
                free( argzz.marked.marked_val );
                marked_len = 0;
        }

        /* init result structure */
        argzz.numMarked = 0;
        argzz.marked.marked_len = 0;
        argzz.marked.marked_val = NULL;

        if (NULL == arg || NULL == arg->objList || arg->numEntries <= 0)
                argzz.status = EP_RB_RECOVER_BAD_ARGS;

        else if ( (argzz.status = check_RPC_state(
                                        FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else if (NULL == (argzz.marked.marked_val =
                                calloc( arg->numEntries, sizeof(bool_t) )) )
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_NO_MEMORY, errno,
                                "Cannot malloc bool_t array" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        else {
                argzz.marked.marked_len = marked_len = arg->numEntries;
                argzz.status = RSTSL_IsObjectMarked( arg->numEntries,
                                                arg->objList,
                                                &argzz.numMarked,
```

```
                                                argzz.marked.
                                                        marked_val );
        }

        set_rpc_obj( re_is_object_marked, &argzz.RPCobjID );

        return &argzz;
}

/*****************************************************************************
**
** Routine:  re_is_object_markable
**
** Inputs:  RE_is_object_markable_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_is_object_markable_result * - result of RPC function call
**
** Purpose:  Function to test if the specified object is markable
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_is_object_markable_result *
re_is_object_markable_1_svc(IN RE_is_object_markable_args *arg,
                        IN struct svc_req *req )
{
        static RE_is_object_markable_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        argzz.markable = FALSE;
        if (NULL == arg || NULL == arg->thisObject)
                argzz.status = EP_RB_RECOVER_BAD_ARGS;

        else if ( (argzz.status = check_RPC_state(
                                        FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else
        {
                argzz.markable = RSTSL_IsObjectMarkable( arg->thisObject );
                argzz.status = E_SUCCESS;
        }

        set_rpc_obj( re_is_object_markable, &argzz.RPCobjID );

        return &argzz;
}

/*****************************************************************************
**
** Routine:  re_find_restorable_objects_1
**
** Inputs:   RE_find_restorable_objects_args * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_find_restorable_objects_result * - result of RPC function call
**
** Purpose:  Function to search for restorable objects in the backup catalog
**
```

```
** Intended caller:  Internal Only.
*************************************************************************
*/

RE_find_restorable_objects_result *
re_find_restorable_objects_1_svc( IN RE_find_restorable_objects_args *arg,
                         IN struct svc_req *req )
{
        static RE_find_restorable_objects_result argzz;
        RE_find_restorable_objects_args               *cmd_args;
        int                                      status;

        setLastRpcTime( );      /* note time of last RPC */

        cmd_args = calloc( 1, sizeof (RE_find_restorable_objects_args) );
        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_NO_MEMORY, errno,
                        "Cannot malloc RE_find_restorable_objects_args"
                                                                    );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no rpc is in progress */
        else if ( (argzz.status = check_RPC_state( TRUE,
                                COMMAND_FIND_RESTORABLE_OBJECTS ))
          != E_SUCCESS )
                ;       /* just return failure status */

        else
        {
                ClearRpcCancelFlag( );  /* reset cancel flag */
                ClearProgressValue( );  /* reset progress count */

                cmd_args->searchCriteria = arg->searchCriteria;
                arg->searchCriteria = NULL;     /* to avoid 2 frees */

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0,
                                                "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand(
                                COMMAND_FIND_RESTORABLE_OBJECTS, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                        status, 0,
                                                "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                        argzz.status = E_SUCCESS;
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_find_restorable_objects_args,
                                (char *)cmd_args );
```

```
                        free( cmd_args );
                }
        }

        set_rpc_obj( re_find_restorable_objects, &argzz.RPCobjID );

        return &argzz;
}

/*************************************************************************
**
** Routine:  re_get_find_results
**
** Inputs:  RE_get_find_results_args  * - args for the RPC call
**
** Outputs:  None
**
** Return Codes:
**          RE_get_find_results_result * - result of RPC function call
**
** Purpose:  Function to retrieve the results of the find_restorable objects
**       request
**
** Intended caller:  Internal Only.
*************************************************************************
*/

RE_get_find_results_result *
re_get_find_results_1_svc(
                IN RE_get_find_results_args *arg, IN struct svc_req *req )
{
        static RE_get_find_results_result       argzz;
        RE_find_restorable_objects_result       *outarg = NULL;
        static RSTRPC_found_obj_list            *last_list = NULL;
        int     result, cmd, status;

        setLastRpcTime( );      /* note time of last RPC */

        if (last_list)
        {       /* free last results */
                xdr_free( xdr_RSTRPC_found_obj_list, (char *)last_list );
                last_list = NULL;
        }

        /* init static output struct */
        argzz.numEntries = 0;
        argzz.cookie = arg->cookie;
        argzz.foundObjs = NULL;

        /* If interrupt was requested, make sure find was running */
        if (arg->interrupt)
        {
                if (E_SUCCESS != (argzz.status =
                    check_RPC_state(
                                FALSE, COMMAND_FIND_RESTORABLE_OBJECTS )))
                {
                /* for get find results after first good get results call: */
                /* find not active -- make sure idle */
                    argzz.status = check_RPC_state (
                                        FALSE, COMMAND_NONE_ACTIVE);
                /* status = E_SUCCESS means call only GetFindResults */
                }

                /* test for completion of find processing: */
                else if (PopResult( 1, &result, &cmd, &status) )
                {
```

```
                if (status == COMMAND_RECORD_GET_FAILED)
                {
                    /* still going: signal cancel, wait till done */
                    SetRpcCancelFlag( );
                    if (PopResult( MAX_CANCEL_WAIT_SECS, &result,
                                &cmd, &status) )
                        /* if no result, error */
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                    else        /* indicate canceled anyway */
                        argzz.status = EP_RB_RECOVER_FIND_INTERRUPTED;
                }
                else {
                    /* log pop error, clean up, return error  */
                    EDMRestoreEng_logent(
                                    __FILE__, __LINE__, LOG_ERR,
                                     status, 0,
                                            "PopResult failed");
                    argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
            }
            else    /* indicate canceled anyway */
                argzz.status = EP_RB_RECOVER_FIND_INTERRUPTED;
    }
    else    /* didn't interrupt -- see if still running: */
    {
            if (E_SUCCESS == (argzz.status =
                check_RPC_state(
                        FALSE, COMMAND_FIND_RESTORABLE_OBJECTS )))
            {
                /* was doing find, test for completion of processing: */
                if (PopResult( 1, &result, &cmd, &status) )
                {
                    if (status == COMMAND_RECORD_GET_FAILED)
                    {   /* not done yet */
                        argzz.numEntries = ReadProgressValue( );
                        argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
                    }
                    else
                    {    /* log error, clean up, return error  */
                        EDMRestoreEng_logent(
                                    __FILE__, __LINE__, LOG_ERR,
                                     status, 0,
                                            "PopResult failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                    }
                }
                else        /* pop worked: indicate results need popping
                                                                        */
                    argzz.status = EP_RB_RECOVER_FIND_INTERRUPTED;
            }
            /* for get find results after first get find results call */
            else if (E_SUCCESS != (argzz.status =
                    check_RPC_state( FALSE, COMMAND_NONE_ACTIVE )))
                    ;       /* another cmd running, inavlid */
            /* argzz.status = E_SUCCESS means call only GetFindResults */
    }

    if (EP_RB_RECOVER_FIND_INTERRUPTED == argzz.status)
    {
    /* popped result, validate and pop output: */
            if (cmd != COMMAND_FIND_RESTORABLE_OBJECTS)
            {
                /* log error, clean up, return error  */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                    MESSAGE_INVALID_COMMAND, 0,
                        "PopResult mismatch: got %d command,
```

```
                                                        expected %d\n",
                                        cmd,
                                          COMMAND_FIND_RESTORABLE_OBJECTS);
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
            }
            else if (result != COMMAND_RESULT_SUCCESS)
            {
               EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                    MESSAGE_FAILURE_DOING_ASYNC_RPC, 0,
                               "RPC failure in process manager thread"
                                                                      );
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
            }
            else if (PopRpcOutput( (void **)&outarg, &status) || (
                                                               !outarg) )
            {
                EDMRestoreEng_logent(
                                    __FILE__, __LINE__, LOG_ERR, status,
                                     0, "PopRpcOutput failure");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
            }
            else
            {
                    argzz.status = outarg->status;  /* get real status */
                    /* free results */
                    xdr_free( xdr_RE_find_restorable_objects_result,
                            (char *)outarg );
                    free( outarg );
                    outarg = NULL;
            }

            clear_RPC_state( );
                                        /* indicate process mgr idle */
    }

    if ( E_SUCCESS == argzz.status
      || EP_RB_RECOVER_FIND_INTERRUPTED == argzz.status )
    {
            /* canceled or done, get some data or free it */
            argzz.status = RSTSL_GetFindResults( arg->interrupt,
                                                  arg->maxEntries,
                                                  &argzz.foundObjs,
                                                  &argzz.numEntries,
                                                  &argzz.cookie );
            last_list = argzz.foundObjs;
    }

    /* return static results struct */
    set_rpc_obj( re_get_find_results, &argzz.RPCobjID );

    if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
        clear_RPC_state( );
                                /* indicate process mgr idle on fatals */

    return &argzz;
}

    /* end of:  re_get_find_results */

/*****************************************************************************
**
** Routine:  re_is_object_searchable
**
** Inputs:  RE_tlo_query_args  * - args for the RPC call
**
** Outputs:  none
**
```

```
** Return Codes:
**          RE_boolean_result *
**
** Purpose:  Function to test if the specified object supports the find api
**
** Intended caller:  Internal Only.
******************************************************************************
*/

RE_boolean_result *
re_is_object_searchable_1_svc(IN RE_tlo_query_args *arg,
                             IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        argzz.boolResult = FALSE;
        if (NULL == arg || NULL == arg->topLevelObj)
                argzz.status = EP_RB_RECOVER_BAD_ARGS;

        else if ( (argzz.status = check_RPC_state(
                                                FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else
        {
                argzz.boolResult = RSTSL_IsObjectSearchable(
                                                        arg->topLevelObj );
                argzz.status = E_SUCCESS;
        }

        set_rpc_obj( re_is_object_searchable, &argzz.RPCobjID );

        return &argzz;
}


/******************************************************************************
**
** Routine:  re_get_backup_times_support
**
** Inputs:  RE_tlo_query_args  * - args for the RPC call
**
** Outputs:  none
**
** Return Codes:
**          RE_boolean_result *
**
** Purpose:  Function to test if the specified object supports restores from
**           multiple backup times
**
** Intended caller:  Internal Only.
******************************************************************************
*/

RE_boolean_result *
re_get_backup_times_support_1_svc( IN RE_tlo_query_args *arg,
                                   IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        argzz.boolResult = FALSE;
        if (NULL == arg || NULL == arg->topLevelObj)
```

```
                argzz.status = EP_RB_RECOVER_BAD_ARGS;

        else if ( (argzz.status = check_RPC_state(
                                                FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else
        {
                argzz.boolResult = RSTSL_GetBackupTimesSupport
                                                (arg->topLevelObj);
                argzz.status = E_SUCCESS;
        }

        set_rpc_obj( re_get_backup_times_support, &argzz.RPCobjID );

        return &argzz;
}


/******************************************************************************
**
** Routine:  re_get_symm_restore_option
**
** Inputs:  RE_tlo_query_args  * - args for the RPC call
**
** Outputs:  none
**
** Return Codes:
**          RE_boolean_result *
**
** Purpose:  Function to test if the specified object supports restores
**           through the Symm
**
** Intended caller:  Internal Only.
******************************************************************************
*/

RE_boolean_result *
re_get_symm_restore_option_1_svc( IN RE_tlo_query_args *arg,
                                  IN struct svc_req *req )
{
        static RE_boolean_result argzz;

        setLastRpcTime( );      /* note time of last RPC */

        argzz.boolResult = FALSE;
        if (NULL == arg || NULL == arg->topLevelObj)
                argzz.status = EP_RB_RECOVER_BAD_ARGS;

        else if ( (argzz.status = check_RPC_state(
                                                FALSE, COMMAND_NONE_ACTIVE ))
          != E_SUCCESS)         /* if not idle, trouble */
                ;       /* we weren't idle, reject call */
        else
        {
                argzz.boolResult = RSTSL_GetSymmRestoreOption
                                                (arg->topLevelObj);
                argzz.status = E_SUCCESS;
        }

        set_rpc_obj( re_get_symm_restore_option, &argzz.RPCobjID );

        return &argzz;
}
```

```
/*****************************************************************************
**
** Routine:  set_rpc_obj
**
** Inputs:   rpc_id      rpc function number
**            rpc_objID  pointer to RPC object ID
**
** Outputs:  None
**
** Return Codes:
**           none
**
** Purpose:  load rpc object id with rpc number and timestamp
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

static void set_rpc_obj( ulong rpc_id, RE_rpc_objID *rpc_objID )
{
        struct timeval timeofday;
        void *dummy = NULL;

        rpc_objID->rpc_type = rpc_id;
        gettimeofday( &timeofday, dummy );
        rpc_objID->time = timeofday.tv_sec;
        return;
}

/*****************************************************************************
**
** Routine:  check_RPC_state
**
**      Function to check if there is any current command, or if it is set to
**      a specific value, and optionally, to set it to a new command value
**
** Inputs:   bool set -          indicates whether this is a request to set
                                                                         the
**                               current command (1/true), or just to check it
**
**           int cmd -           if set input is 0/false,
                                                   command value to check
**                               for (COMMAND_NONE_ACTIVE means idle)
**                               if set is 1/true, value to change current
**                               command to,
                                           after verifying that is it not set,
**                               i.e., that it is set to COMMAND_NONE_ACTIVE.
**
** Outputs:  None
**
** Return Codes:
**           RE_errno_ty result  - result of check: E_SUCCESS if current
**                                   command was in desired state;
**                                   EP_RB_RECOVER_INVALOP otherwise
**
** Purpose:  verify that no async RPC is actice,
                                       or that specified one IS active
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

static RE_errno_ty check_RPC_state( boolean_ty set, int cmd )
{
        if ( (!set && cmd != current_rpc_cmd)
          || (set && current_rpc_cmd != COMMAND_NONE_ACTIVE))
```

```
                /* check-only failure or can't set because another RPC busy
                                                                           */
                return EP_RB_RECOVER_INVALOP;
        else {  /* check succeeded, change current cmd if requested */
                if (set)
                        current_rpc_cmd = cmd;
                return E_SUCCESS;
        }
}

/*****************************************************************************
**
** Routine:  clear_RPC_state
**
**      Function to clear the current RPC command
**
** Inputs:   none
**
** Outputs:  None
**
** Return Codes:         none
**
** Purpose:  indicate that no async RPC is active
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

static void clear_RPC_state(
                                void )            /* indicate process mgr idle */
{
        current_rpc_cmd = COMMAND_NONE_ACTIVE;
}

/*****************************************************************************
**
** Routine: re_load_recx_directives
**
** Inputs:  RE_recx_file_info  * - args for the RPC call to get directives
                                                                        file
**
** Outputs: RE_status_result * - result of RPC function call
**
** Purpose:  Function to load the recx file into the recx struct and then
**           into context structure
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_status_result *
re_load_recx_directives_1_svc( IN RE_recx_file_info *arg,
                                  IN struct svc_req *req )
{
        static RE_status_result          argzz;
       RSTRPC_recx_file_info       *fileinfo;
        RSTRPC_recx_file_info       *cmd_args;
        int                         status;
        cmd_args = calloc( 1, sizeof( RSTRPC_recx_file_info) );

        fileinfo = &arg->fileinfo;

        if (NULL == cmd_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_NO_MEMORY, errno,
                     "Cannot malloc RE_recx_file_info structure" );
                argzz.status = EP_RB_RECOVER_NOMEM;
        }
        /* make sure no rpc is in progress */
        else if ( (argzz.status = check_RPC_state( TRUE,
                  COMMAND_LOAD_RECX_DIRECTIVES )) != E_SUCCESS )
                ;       /* just return failure status */

        else
        {
                ClearRpcCancelFlag( );  /* reset cancel flag */
                ClearProgressValue( );  /* reset progress count */

                cmd_args->hostname = esl_strdup( fileinfo->hostname );
                cmd_args->filename = esl_strdup( fileinfo->filename );

                if (PushRpcInput( (void *)cmd_args, &status) )
                {
                        /* log error, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                   "PushRpcInput failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else if (PushCommand(
                                   COMMAND_LOAD_RECX_DIRECTIVES, &status) )
                {
                /* log error, clean up input queue, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0,
                                                   "PushCommand failed");
                        PopRpcInput( (void **)&cmd_args, &status);
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                        clear_RPC_state( );     /* indicate idle on fatals */
                }
                else
                  {
                        argzz.status = E_SUCCESS;
                  }
        }

        if (argzz.status != E_SUCCESS)
        {       /* failure somewhere: free allocated memeory: */
                if (cmd_args) {
                        xdr_free( xdr_RE_recx_file_info, (char *)cmd_args );
                        free( cmd_args );
                }
        }

    set_rpc_obj( re_poll_load_recx_directives, &argzz.RPCobjID );

    return &argzz;
}

/*****************************************************************************
**
** Routine:  re_poll_load_recx_directives_1_svc
**
** Inputs:   RE_null_args
**
** Outputs:  RE_status_result
**
** Purpose:  Function to test for completion of the previously started
**            RE_load_recx_directives operation.
```

```
**
** Intended caller:  Internal Only.
*****************************************************************************
*/

RE_status_result *
re_poll_load_recx_directives_1_svc( IN RE_null_args *arg,
                                 IN struct svc_req *req )
{
        static RE_status_result         argzz;
        static RE_status_result         *outarg = NULL;
        int     result, cmd, status;

        if (outarg)
        {       /* free last results */
                xdr_free( xdr_RE_status_result, (char *)outarg );
                free( outarg );
                outarg = NULL;
        }

        /* make sure submit is in progress */
        if ( (argzz.status = check_RPC_state(
                                  FALSE, COMMAND_LOAD_RECX_DIRECTIVES ))
          != E_SUCCESS )
                ;       /* just return failure status */
        /* test for completion of processing: later use real flag */
        else if (PopResult( -1, &result, &cmd, &status) )
        {
                if (status == COMMAND_RECORD_GET_FAILED)
                {
                        argzz.status = EP_RB_RECOVER_RPC_INCOMPLETE;
               }

                else {
                        /* log error, clean up, return error  */
                        EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                              status, 0, "PopResult failed");
                        argzz.status = EP_RB_RECOVER_SERVERFAIL;
                }
        }

        if (argzz.status != E_SUCCESS)
                ;       /* fall thru to error return logic */

        else if (cmd != COMMAND_LOAD_RECX_DIRECTIVES)
        {
                /* log error, clean up, return error  */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_INVALID_COMMAND, 0,
                        "PopResult mismatch: got %d command, expected %d\n",
                                      cmd, COMMAND_LOAD_RECX_DIRECTIVES);
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (result != COMMAND_RESULT_SUCCESS)
        {
              EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                   MESSAGE_FAILURE_DOING_ASYNC_RPC, 0,
                              "RPC failure in process manager thread" );
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else if (PopRpcOutput( (void **)&outarg, &status) )
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR, status,
                                      0, "PopRpcOutput failure");
                argzz.status = EP_RB_RECOVER_SERVERFAIL;
        }
        else
        {       /* return popped results struct */
            set_rpc_obj( re_poll_load_recx_directives, &outarg->RPCobjID);
            clear_RPC_state( );
                                        /* indicate process mgr idle */
            return outarg;
        }

        set_rpc_obj( re_poll_load_recx_directives, &argzz.RPCobjID );
        if (argzz.status == EP_RB_RECOVER_SERVERFAIL)
            clear_RPC_state( );
                                /* indicate process mgr idle on fatals */

        return &argzz;
}



/******************************************************************
 *  RE_get_catalog_info:
 *
 *  This routine returns the level structure with the
 *  level for backup being restored
 *
 * OutPuts:
 *        RE_catalog_info Struct containing The level of the backup,
 *        the number of records, and the catalog type for the backup
 *         being restored
 * Parameters:
 *    RE_time *arg     (I) Time of the backup that is beeing looked at
 *
 * Return Codes: (Stored in argzz.status)
 *             EP_RB_RECOVER_RPC_FAIL - if rpc call failed because the
 *                                      argument was NULL
 *             E_SUCCESS              - if rpc call completed successfully
 *             EP_RB_RECOVER_INVALOP  - if another RPC is running
 *                                      this result is gotten from
 *                                      check_rpc_state
 ******************************************************************/

RE_catalog_info *
re_get_catalog_info_1_svc( IN RE_time *arg,
                           IN struct svc_req *req )
{
        static RE_catalog_info    argzz;
                                      /* variable to return to RPC caller*/
         if (
       NULL == arg)  /* we need the input to continue, so if none passed in */
                argzz.status = EP_RB_RECOVER_RPC_FAIL;

         else if ( (argzz.status = check_RPC_state(
                                               FALSE, COMMAND_NONE_ACTIVE ))
           != E_SUCCESS)         /* if RPC not idle, trouble */
                ;       /* we weren't idle, reject call */
         else {  /*we are ok to run an RPC :) */
                /* Call the Function to get the catalog info and place
                 * its result in the status of the return struct.
                 * this call should fill in the required fields
                 */
                argzz.status = RSTSL_get_catalog_info( arg->backupTime,
                                                       &argzz.level,
                                                       &argzz.numrec,
                                                       &argzz.catType);
         }
```

```
                set_rpc_obj( re_get_catalog_info, &argzz.RPCobjID );

                return &argzz;
                        /* return our newly retrieved values from the catalog*/
}
```

```
/*************************************************************************
 **
 ** File Name:   RSTinitfin.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **               This module contains the Restore API functions to
 **               initialize and terminate the restore operation.
 **
 ** Table of Contents:
 ** -------------------
 **       API Functions:
 **               EDMRST_Initialize
 **               EDMRST_Finish
 **
 **       Internal Functions:
 **
 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
 ************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1

/*
 * System headers.
 */
#include <pwd.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>

/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>
```

```
/*
 * Comms headers.
 */
#include <restore/csc_EDMDispatch.h>
#include <restore/csc_EDMRestoreEng.h>
#include <restore/dispatch_daemon.h>
#include <restore/restore_engine.h>
#include <edmlink/edmlink_api.h>

/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * Global declarations
 */

   internalHandlePtr handlePtr = NULL;
```

```
/**********************************************************************
 * EDMRST_Initialize:
 *
 *  This function takes care of all the initialization for a recovery
 *  session.  This must be called prior to any of the other functions
 *  in the Recover API.
 *
 * Parameters:
 *
 *  hostname    (I) - The machine name of the server to use.
 *  svrHdl      (O) - A handle to receive a pointer to this user's client
 *                    handle for the Restore Engine connection.
 *  timeout     (I) - The maximum number of seconds to wait for the connection
 *                    to the Restore Engine process to be completed.
 *
 **********************************************************************/

eerrno_ty
EDMRST_Initialize( hostname_ty   hostname,
                   serverHandle  *svrHdl,
                   unsigned long timeout )
{
        eerrno_ty api_status = E_SUCCESS;

        uid_t   human_uid;
        struct  passwd *pw;
        char    *human_uidname ;

        RE_initialize_args      re_init_args;
        RE_status_result        *re_init_result;
        rpc_if_handle_t                 re_if_spec;
        rpc_binding_handle_t    re_handle;
        int                     retval;
        time_t                  end_time;

#ifdef DEBUG
#define  RPC_TIMEOUT     3600
        struct timeval  rpc_timeout;
#endif

/********************** BEGINNING OF Dispatch Daemon STUFF ***************/
    error_status_t status;
    DD_initialize_args    initargs;
    DD_getservicestatus_args  statargs;
    DD_initialize_result  *initres = NULL;
    DD_getservicestatus_result  *statres = NULL;
    rpc_if_handle_t       if_spec;


    time( &end_time );   /* compute time to give up waiting */
    end_time += timeout;

    memset(&if_spec,0,sizeof(rpc_if_handle_t));
    memset(&re_if_spec,0,sizeof(rpc_if_handle_t));

    if (svrHdl == NULL || hostname == NULL )
    {
        return( EP_RB_RECOVER_BAD_ARGS );
    }

         rec_api_log_begin( "edmrestore_api" );
                                              /* init logs, ignore errs?? */

         /* get user name to pass to DD and RE */
         human_uid = getuid();
         pw = getpwuid( human_uid );
```

```
         if (pw == NULL || NULL == pw->pw_name )
         {
                 /* Trouble.  */

                 rec_api_log_csm(SUB_CSM_USER_NOT_IN_PASSWD, NULL);
                 return(EP_RB_RECOVER_PERMISSION_DENIED);
         }

         human_uidname = pw->pw_name;

    handlePtr = (internalHandle *) calloc(1, sizeof(internalHandle));

    /* Use this macro to setup the interface spec */
    CLIENT_IFSPEC(if_spec);

    /* Arrive at a server binding.  Note that if they didn't give us
    ** a valid host parameter, this will fail and drop through and
    ** return NULL in the end.
    ** This call will get and store a fully resolved binding
    ** handle to the host.  The first time we ever call the host,
    ** csc_get_handle will resolve and store the binding.  If we
    ** ever use csc_get_handle to talk to the same host again,
    ** it will just give back the previously resolved binding.
    */

    retval = csc_get_handle((unsigned char *) hostname,
                            if_spec,
                            SERVER_GROUP,
                            &handlePtr -> dd_binding_handle,
                            &status);

    /*
    ** Find out if we got csc handle and see if status is bad.
    ** error_status_ok is a macro defined in cscomm.h.
    */

    if ((status != error_status_ok) || (retval == 0))
    {
      /*  If errno not set, use status if it is a valid errno value */

      if ( errno == 0 )
        errno = ( strerror( status ) ? status : ETIME );

      rec_api_log_csm( SUB_CSM_RPC_FAIL,
                       "failure finding edmdispd to start restore engine" );

      return EP_RB_RECOVER_SERVERFAIL;
    }

    errno = 0;

#ifdef DEBUG
/* increase rpc timeout during debugging */
    rpc_timeout.tv_sec = RPC_TIMEOUT;
    rpc_timeout.tv_usec = 0;
    clnt_control( handlePtr->dd_binding_handle, CLSET_TIMEOUT,
                  (char *)&rpc_timeout );
#endif

    initargs.service = DD_SERVICE_RESTORE;
    initargs.hostname = hostname;
    initargs.username = human_uidname;
    initargs.timeout = timeout;

    initres = dd_initialize_1( &initargs, handlePtr -> dd_binding_handle );
                /* Will have _1 for RPC call  */
    if (initres == NULL)
    {
        return EP_RB_RECOVER_RPC_FAIL;
    }

    statargs.service_handle = initres -> service_handle;
    statargs.status = 0;

    statres = dd_getservicestatus_1( &statargs, handlePtr->dd_binding_handle );

    if (statres == NULL)
    {
        return EP_RB_RECOVER_RPC_FAIL;
    }

    while (statres -> status == DD_SERVICE_STARTING )
    {
         time_t  now;

         xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
         time( &now );
         if (now >= end_time)
         {
                 rec_api_log_csm( SUB_CSM_RPC_FAIL,
                        "timeout waiting for edmdispd to start restore engine"
                                                                            );
                 return EP_RB_RECOVER_SERVERFAIL;
         }

        sleep(1);

        statres = dd_getservicestatus_1( &statargs,
                                         handlePtr -> dd_binding_handle );

        if (statres == NULL)
        {
             rec_api_log_csm( SUB_CSM_RPC_FAIL,
       "failure getting status from edmdispd while starting restore engine" );
            return EP_RB_RECOVER_RPC_FAIL;
        }
    }

    if (statres -> status != DD_SERVICE_RUNNING)
    {
         rec_api_log_csm( SUB_CSM_RPC_FAIL,
                      "edmdispd failure while starting restore engine" );
         xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
        return EP_RB_RECOVER_SERVERFAIL;
    }

    memcpy( handlePtr -> opaque128,
            statres -> handle.handle_val,
            sizeof(handlePtr -> opaque128) );

    xdr_free( xdr_DD_getservicestatus_result, (char *)statres );

/********************** END OF Dispatch Daemon STUFF ***************/

/* Restore Engine FUNCTIONALITY BEGINS HERE */

/*      RE_CLIENT_IFSPEC(re_if_spec);  */

        retval = csc_private_ifspec_init(
                              (unsigned char *) handlePtr -> opaque128,
                             RE_PROGNUM,
```

```
                             RE_VERSNUM,
                              &re_if_spec,
                              &status);

        if (retval == 0)
        {
             rec_api_log_csm( SUB_CSM_RPC_FAIL,
                         "failure initializing interface to restore engine"
                                                                          );
            return EP_RB_RECOVER_SERVERFAIL;
        }

         api_status = EP_RB_RECOVER_SERVERFAIL;
         do {
                 time_t  now;
                 time( &now );
                 if (now >= end_time)
                 {
                     rec_api_log_csm( SUB_CSM_RPC_FAIL,
                                 "timeout connecting to restore engine" );
                     return EP_RB_RECOVER_SERVERFAIL;
                 }

                 sleep( 1 );     /* give restore engine time to get going */
                 retval = csc_connect_to_rpc_service(
                                        (unsigned char *)hostname,
                                        re_if_spec,
                                        RE_CLIENT_GROUP,
                                        &handlePtr -> re_binding_handle,
                                        &status);
                 if ((status == error_status_ok) && (retval != 0))
                         api_status = E_SUCCESS;
        } while (api_status != E_SUCCESS);

        if (api_status == E_SUCCESS)
        {
                re_handle = handlePtr -> re_binding_handle;
#ifdef DEBUG
/*      increase rpc timeout during debugging */
                rpc_timeout.tv_sec = RPC_TIMEOUT;
                rpc_timeout.tv_usec = 0;
                clnt_control( re_handle, CLSET_TIMEOUT, (
                                                    char *)&rpc_timeout );
#endif
                re_init_args.username = human_uidname;
                set_rpc_obj( re_initialize, &re_init_args.RPCobjID );
                re_init_result = re_initialize_1( &re_init_args, re_handle );
                if (!re_init_result) {
                        api_status = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL,
                             "failure communicating with restore engine" );
                }
                else {
                        api_status = re_init_result->status;
                        /* release RPC result struct: */
                        xdr_free( xdr_RE_status_result, (
                                                   char *)re_init_result);
                }
        }

        else
            rec_api_log_csm( SUB_CSM_RPC_FAIL,
                            "failure connecting to restore engine" );

        if (
          api_status == E_SUCCESS)    /* return rest eng handle on success */
```

```
                *svrHdl = (serverHandle)re_handle;

        return( api_status );

}       /* End of EDMRST_Initialize() */
```

```
/*******************************************************************
 * Ping:
 *
 *  This function allows a ping to be issued in order to keep the
 *  engine alive and running so that the engine will not time out.
 *
 * Parameters:
 *
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *
 *******************************************************************/
eerrno_ty EDMRST_Ping( serverHandle svrHdl )
{
  eerrno_ty      api_status = E_SUCCESS;
  RE_null_args   re_ping_args;
  RE_status_result       *re_ping_result = NULL;


  if ( NULL == svrHdl || NULL == handlePtr
       || svrHdl != handlePtr->re_binding_handle )
    {
      return( EP_RB_RECOVER_BAD_ARGS );
    }

  set_rpc_obj( re_ping, &re_ping_args.RPCobjID );
  re_ping_result = re_ping_1( &re_ping_args, svrHdl );
  if (NULL == re_ping_result) {
    api_status = EP_RB_RECOVER_RPC_FAIL;
    rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
  }
  else {
    api_status = re_ping_result->status;
    /* release RPC result struct: */
    xdr_free( xdr_RE_status_result, (char *)re_ping_result);
  }

}



/*******************************************************************
 * EDMRST_Finish
 *
 *    Function Description:
 *
 *  This function terminates a restoral session,
                                                    but only during the browse and
 *  mark phase. It will be rejected if a restore is currently being executed.
 *  This routine will clean up any local memory used in the session and will
 *  disconnect from the Restore Engine.  After calling this function,
 *  EDMRST_Initialize MUST be called before calling any other functions in
                                                                            this
 *  API.
 *
 * Parameters:
 *
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *
 *    Return Codes:
 *       EP_RB_RECOVER_BAD_ARGS
 *       EP_RB_RECOVER_RPC_FAIL
 *       EP_RB_RECOVER_INVALOP
 *       EP_RB_RECOVER_SERVERFAIL
 *
 */

eerrno_ty
EDMRST_Finish( serverHandle svrHdl )
{
        eerrno_ty       api_status = E_SUCCESS;
        RE_null_args    re_finish_args;
        RE_status_result        *re_finish_result = NULL;
        int             csc_status;

        if ( NULL == svrHdl || NULL == handlePtr
          || svrHdl != handlePtr->re_binding_handle )
        {
                return( EP_RB_RECOVER_BAD_ARGS );
        }

        set_rpc_obj( re_finish, &re_finish_args.RPCobjID );
        re_finish_result = re_finish_1( &re_finish_args, svrHdl );
        if (!re_finish_result) {
                api_status = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                api_status = re_finish_result->status;
                /* release RPC result struct: */
                xdr_free( xdr_RE_status_result, (char *)re_finish_result);
        }

        rec_api_log_end();      /* write last log and close the log file. */


        return( api_status );

}       /* EDMRST_Finish */
```

```
/***************************************************************************
 **
 ** File Name:   RSTgettlob.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **       This module contains the EDMRST_GetTopLevelObjects
 **       Restore API function.
 **       This function is provided to allow retrieval of the
 **       top level objects which are restorable for the given client.
 **
 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
 ***************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_rpc.h>
#include <RSTsup_csm.h>


/*
 * External declarations
 */
```

```
/*****************************************************************
 * EDMRST_GetTopLevelObjects:
 *
 *  This function is provided to allow retrieval of the
 *  work items which are restorable for the given client.
 *
 *  It is a GOAL of this routine to return all work-items ever backed
 *  up successfully.  Currently, though, it only looks in the config
 *  file for work-items of the given client.
 *
 *  The cookie must be initialized to INIT_COOKIE on the first call to this
 *  routine. This routine will update the cookie to allow retrieval of more
 *  objects if there are more than "maxEntries".  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * Parameters:
 *   svrHdl         (I)  - A pointer to this user's client handle for the
 *                          Restore Engine (server) connection.
 *   sourceHost     (I)  - the name of the source host being restored
 *   maxEntries     (I)  - the maximum number of objects to return
 *   topLevObjs     (O)  - ptr to pre-allocated array of restorableObject
 *                          buffer ptrs
 *   numberEntries (O)  - the real number of objects returned in the array
 *   cookie         (IO) - a place holder for the list position
 *                          meaningful to only the internals of the API
 *
 *****************************************************************/

eerrno_ty
EDMRST_GetTopLevelObjects( serverHandle        svrHdl,
                           const char          *sourceHost,
                           const short         maxEntries,
                           restorableObjectPtr *topLevObjs,
                           short               *numberEntries,
                           long                *cookie )
{
        RE_get_top_level_objects_result         *rpc_result;
        RE_get_top_level_objects_args   rpc_args;
        RSTRPC_tlo_list                         *temp_list;
        eerrno_ty                       result = E_SUCCESS ;
        short                           index;
        restorableObject        **objPtrArray = (
                                        restorableObject **)topLevObjs;

        rbe_log_debug_sub( 0, "EDMRST_GetTopLevelObjects called" );

        /* validate args first: */
        if (sourceHost==NULL
         || numberEntries==NULL
         || cookie==NULL
         || maxEntries <= 0
         || topLevObjs== NULL
         || svrHdl==NULL )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* validate target restorableObjects: */
        for ( index=0; index<maxEntries; index++ )
        {
                if (RESTORABLE_OBJECT != objPtrArray[index]->restoreObjType
                  || NULL != objPtrArray[index]->rpcObjPtr )
                return( EP_RB_RECOVER_BAD_ARGS );
        }
        /* Prepare input argument structure for RPC: */
        rpc_args.sourceHost = (char *)sourceHost;
        rpc_args.maxEntries = maxEntries;
        rpc_args.cookie = *cookie;
```

```
        set_rpc_obj( re_get_top_level_objects, &rpc_args.RPCobjID );

        rpc_result = re_get_top_level_objects_1( &rpc_args, svrHdl );
        if (!rpc_result) {
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                return( EP_RB_RECOVER_RPC_FAIL );
        }

        /* move results to caller's area, if successful: */
        if (rpc_result->status == E_SUCCESS)
        {
                *cookie = rpc_result->cookie;
                *numberEntries = rpc_result->numEntries;
                index = 0;
                while ( rpc_result->numEntries )
                {
                        temp_list = rpc_result->topLevelObjs;
                        if ( !temp_list || !rpc_args.maxEntries-- )
                                /* some null pointer or too many returned */
                                break;
                        objPtrArray[index++]->rpcObjPtr
                                = (RSTRPC_restorable_obj_root *)temp_list->tlo;
                /* need this to end with NULL in rpc_result->topLevelObjs,
                 * because returned top level objects can't be freed yet */
                        rpc_result->topLevelObjs = temp_list->next;
                        free( temp_list );
                        rpc_result->numEntries--;
                }
                if (rpc_result->numEntries)
                        rpc_result->status = EP_RB_RECOVER_SERVERFAIL;
        }

        result = rpc_result->status;

        /* release RPC result struct: */
        xdr_free( xdr_RE_get_top_level_objects_result, (char *)rpc_result );

        return( result );
}       /* end of EDMRST_GetTopLevelObjects() */


/*****************************************************************
 * EDMRST_GetAllTopLevelObjects:
 *
 *  This function is provided to allow retrieval of the
 *  work items which are restorable for the given client.
 *
 *  It is a GOAL of this routine to return all work-items ever backed
 *  up successfully.  Currently, though, it only looks in the config
 *  file for work-items of the given client.
 *
 *  The cookie must be initialized to INIT_COOKIE on the first call to this
 *  routine. This routine will update the cookie to allow retrieval of more
 *  objects if there are more than "maxEntries".  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * Parameters:
 *   svrHdl         (I)  - A pointer to this user's client handle for the
 *                          Restore Engine (server) connection.
 *   sourceHost     (I)  - the name of the source host being restored
 *   maxEntries     (I)  - the maximum number of objects to return
 *   topLevObjs     (O)  - ptr to pre-allocated array of restorableObject
 *                          buffer ptrs
 *   numberEntries (O)  - the real number of objects returned in the array
 *   cookie         (IO) - a place holder for the list position
 *                           meaningful to only the internals of the API
 *
 ****************************************************************/

eerrno_ty
EDMRST_GetAllTopLevelObjects( serverHandle        svrHdl,
                            const char          *sourceHost,
                             const short         maxEntries,
                             restorableObjectPtr *topLevObjs,
                             short               *numberEntries,
                             long                *cookie )
{
        RE_get_top_level_objects_result         *rpc_result;
        RE_get_top_level_objects_args   rpc_args;
        RSTRPC_tlo_list                         *temp_list;
        eerrno_ty                       result = E_SUCCESS ;
        short                           index;
        restorableObject        **objPtrArray = (
                                        restorableObject **)topLevObjs;

        rbe_log_debug_sub( 0, "EDMRST_GetAllTopLevelObjects called" );

        /* validate args first: */
        if (sourceHost==NULL
         || numberEntries==NULL
         || cookie==NULL
         || maxEntries <= 0
         || topLevObjs== NULL
         || svrHdl==NULL )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* validate target restorableObjects: */
        for ( index=0; index<maxEntries; index++ )
        {
                if (RESTORABLE_OBJECT != objPtrArray[index]->restoreObjType
                  || NULL != objPtrArray[index]->rpcObjPtr )
                return( EP_RB_RECOVER_BAD_ARGS );
        }
        /* Prepare input argument structure for RPC: */
        rpc_args.sourceHost = (char *)sourceHost;
        rpc_args.maxEntries = maxEntries;
        rpc_args.cookie = *cookie;
        set_rpc_obj( re_get_all_top_level_objects, &rpc_args.RPCobjID );

        rpc_result = re_get_all_top_level_objects_1( &rpc_args, svrHdl );
        if (!rpc_result) {
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                return( EP_RB_RECOVER_RPC_FAIL );
        }

        /* move results to caller's area, if successful: */
        if (rpc_result->status == E_SUCCESS)
        {
                *cookie = rpc_result->cookie;
                *numberEntries = rpc_result->numEntries;
                index = 0;
                while ( rpc_result->numEntries )
                {
                        temp_list = rpc_result->topLevelObjs;
                        if ( !temp_list || !rpc_args.maxEntries-- )
                                /* some null pointer or too many returned */
                                break;
                        objPtrArray[index++]->rpcObjPtr
                            = (RSTRPC_restorable_obj_root *)temp_list->tlo;
                /* need this to end with NULL in rpc_result->topLevelObjs,
                 * because returned top level objects can't be freed yet */
```

```
                        rpc_result->topLevelObjs = temp_list->next;
                        free( temp_list );
                        rpc_result->numEntries--;
                }
                if (rpc_result->numEntries)
                        rpc_result->status = EP_RB_RECOVER_SERVERFAIL;
        }

        result = rpc_result->status;

        /* release RPC result struct: */
        xdr_free( xdr_RE_get_top_level_objects_result, (char *)rpc_result );

        return( result );
}       /* end of EDMRST_GetNetworkTopLevelObjects() */
```

```
/***************************************************************************
 **
 ** File Name:   RSTgetrobs.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **
 ** This module contains the EDMRST_GetRestorableObjects API and some of its
 ** support functions.
 **
 ** Table of Contents:
 ** ------------------
 **       API Functions:
 **
 **              EDMRST_GetRestorableObjects
 **
 **
 ** Compile-Time Options:
 **
 ***************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */
#include <unistd.h>      /* doesn't help define usleep(), so... */
extern int usleep( unsigned int );


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>
```

```
/*
 * #defines, structures, typedefs local to this source file
 */

#define RST_MAX_GET_ROBJS_DELAY 3        /* max seconds between polls */

/*
 * External declarations
 */

NEW_SRC_FILE();


/*
 * Local function prototypes
 */
```

```
/*********************************************************************
 * EDMRST_GetRestorableObjects API
 *
 *    Function Description:
 *
 *  Given a parent (i.e. container) restorable object, return the children
 *  restorable objects it contains.
 *
 *  The cookie must be initialize to INIT_COOKIE on the first call to this
 *  routine. This routine will update the cookie to allow retrieval of more
 *  objects if there is more than "maxEntries".  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 *    Parameters:
 *      svrHdl          - (I) A pointer to this user's client handle for the
 *                            Restore Engine (server) connection.
 *      parentPtr       - (I) ptr to parent restorableObject
 *      allowBF               - (
                          I) flag indicating whether or not to include bad
 *                            files
 *      maxEntries      - (I) max. # of entries that the preallocated buffer
 *                          can hold
 *      objBufPtr       - (O) ptr to preallocated array of restorableObject
 *                          buffer ptrs
 *      numEntries      - (O) ptr to buffer to receive number of entries
 *                          returned in objBufArray
 *      cookie          - (
                        I/O) ptr to a long integer whose value is meaningful
 *                             to only the internals of the API
 *
 *                             == INIT_COOKIE: initial query input
 *                             == DONE_COOKIE: end of query output
 *
 *  Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *
 *      EP_RB_RECOVER_INVAL_OBJNAME     - input restorableObject does
 *                                        not have a valid name;
 *      EP_RB_RECOVER_BAD_ARGS          - input restorableObject ptr
 *                                        or objBufPtr is NULL;
 *      EP_RB_RECOVER_BAD_COOKIE        - input cookie ptr is NULL or
 *                                        the cookie is DONE_COOKIE;
 *      EP_RB_RECOVER_INVALOP           - the call is issued without
 *                                        the correct context setup;
 *
 *      return codes from GetWorkItemContents() and from
 *      GetDirContents().
 */

eerrno_ty
EDMRST_GetRestorableObjects( serverHandle       svrHdl,
                            const restorableObjectPtr parentPtr,
                             const boolean_ty    allowBF,
                             const long                maxEntries,
                             restorableObjectPtr *objBufPtr,
                             long                *numEntries,
                             long                 *cookie )
{
        RE_get_restorable_objects_start_result  *start_rpc_result = NULL;
        RE_get_restorable_objects_start_args    start_rpc_args;
        RE_get_restorable_objects_output_result         *output_rpc_result =
                                                                      NULL;
        RE_get_restorable_objects_output_args   output_rpc_args;
        RSTRPC_uro_list                         *temp_list;
        struct RSTRPC_restorable_obj_root       *temp_robj;
        eerrno_ty                         result = E_SUCCESS ;
```

```
        short                           index;
        restorableObject        **objPtrArray = (
                                                restorableObject **)objBufPtr;

        rbe_log_debug_sub( 0, "EDMRST_GetRestorableObjects called" );

        /* validate args first: */
        if (parentPtr==NULL
          || numEntries==NULL
          || cookie==NULL
          || maxEntries <= 0
          || objBufPtr== NULL
          || svrHdl==NULL )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* validate target restorableObjects: */
        for ( index=0; index<maxEntries; index++ )
        {
                if ( (NULL == objPtrArray[index])
                   || (
                       RESTORABLE_OBJECT != objPtrArray[index]->restoreObjType)
                   || (NULL != objPtrArray[index]->rpcObjPtr )
                )
                        return( EP_RB_RECOVER_BAD_ARGS );
        }

        if (*cookie == DONE_COOKIE)
                return(EP_RB_RECOVER_BAD_COOKIE);

        /* validate parent object type as top level or container */
        if (NULL == (temp_robj = ((restorableObject *)parentPtr)->rpcObjPtr))
                return( EP_RB_RECOVER_BAD_ARGS );

        if (RESTORABLE_OBJECT !=
            ((restorableObject *)parentPtr)->restoreObjType )
                return EP_RB_RECOVER_INVAL_OBJTYPE;

        if ( (temp_robj->objLevel != RSTRPC_tlo_type)
          && (temp_robj->objLevel != RSTRPC_container_type) )
        {
                if (temp_robj->objLevel != RSTRPC_leaf_type)
                        return( EP_RB_RECOVER_INVAL_OBJTYPE );
                else
                        return( EP_RB_RECOVER_INVALOP );
        }

        /* request the restorable objects with one RPC */

        /* Prepare input argument structure for RPC: */
        start_rpc_args.parentObj = malloc( sizeof(struct RE_restorable_obj) );
        start_rpc_args.parentObj->objLevel = temp_robj->objLevel;
        start_rpc_args.parentObj->RE_restorable_obj_u.uroInfo
                = (struct RSTRPC_user_restorable_object *)temp_robj;
        start_rpc_args.allowBadFiles = allowBF;
        start_rpc_args.cookie = *cookie;
        start_rpc_args.maxEntries = maxEntries;
        set_rpc_obj( re_get_restorable_objects_start,
                     &start_rpc_args.RPCobjID );

        start_rpc_result = re_get_restorable_objects_start_1(
                                                         &start_rpc_args,
                                                           svrHdl );

        if (!start_rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
```

```
	else
	{
		result = start_rpc_result->status;

		/* release RPC result struct: contents and struct */
		xdr_free( xdr_RE_get_restorable_objects_start_result,
			(char *)start_rpc_result );
	}

	free( start_rpc_args.parentObj );

	/* prepare to call another RPC for results, if successful: */
	if (result != E_SUCCESS)
		return( result );
	else
		result = EP_RB_RECOVER_RPC_INCOMPLETE;

	output_rpc_args.maxEntries = maxEntries;

	/* poll for completion or error */
	while (result == EP_RB_RECOVER_RPC_INCOMPLETE)
	{
		unsigned int poll_delay = 100000;	/* .1 second */
		set_rpc_obj( re_get_restorable_objects_output,
			&output_rpc_args.RPCobjID );
		output_rpc_result = re_get_restorable_objects_output_1(
						&output_rpc_args,
						svrHdl );
		if (!output_rpc_result) {
			result = EP_RB_RECOVER_RPC_FAIL;
			rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
		}
		else
			result = output_rpc_result->status;

		if (result == EP_RB_RECOVER_RPC_INCOMPLETE)
		{
			/* release RPC result struct: contents and struct */
			xdr_free(
				xdr_RE_get_restorable_objects_output_result,
				 (char *)output_rpc_result );
			output_rpc_result = NULL;
			/* wait till next poll */
			usleep( poll_delay );
			if (poll_delay < RST_MAX_GET_ROBJS_DELAY) {
			    poll_delay *= 2;
			    if (poll_delay > RST_MAX_GET_ROBJS_DELAY)
				poll_delay = RST_MAX_GET_ROBJS_DELAY;
			}
		}
	}

	/* move results to caller's area, if successful: */
	if (result == E_SUCCESS)
	{
		*cookie = output_rpc_result->cookie;
		*numEntries = output_rpc_result->numEntries;
		index = 0;
		while ( output_rpc_result->numEntries )
		{
			temp_list = output_rpc_result->childrenObjs;
			if ( !temp_list || !output_rpc_args.maxEntries-- )
				break;
				/* null pointer or too many returned */
			objPtrArray[index]->rpcObjPtr =
			    (RSTRPC_restorable_obj_root *)temp_list->uro;
```

```
			/* needed to end with NULL in output_rpc_result->childrenObjs,
			 * because returned user rest. objects can't be freed yet */
				output_rpc_result->childrenObjs = temp_list->next;
				free( temp_list );
				output_rpc_result->numEntries--;
				index++;
			}
			if (output_rpc_result->numEntries)
				result = EP_RB_RECOVER_SERVERFAIL;
		}

		/* release RPC result struct's contents and itself: */
		if (output_rpc_result) {
			xdr_free( xdr_RE_get_restorable_objects_output_result,
				(char *)output_rpc_result );
		}

		return( result );
}		/* EDMRST_GetRestorableObjects */
```

```
/*************************************************************************
 **
 ** File Name:   RSTgtbkups.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **
 ** This module contains the Restore API functions that set the
 ** recover_context to a specific time of the backup and a number
 ** of query functions against the currently setup backup.
 **
 ** Table of Contents:
 ** -------------------
 **        EDMRST_SetPrevBackup
 **        EDMRST_SetNextBackup
 **        EDMRST_SetFirstBackup
 **        EDMRST_SetMostRecentBackup
 **        EDMRST_SetBackupForTime
 **        EDMRST_GetCurrentBackupTime
 **        EDMRST_GetCurrentTemplate
 **        EDMRST_GetAllBackupTimes
 **
 **
 ** Compile-Time Options:
 **               This section must list any compile time definitions
 **               which will affect this header.
 **
 *************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif

#define  _POSIX_SOURCE 1


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */


/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <ebutil/ebutil.h>
#include <eb/rb_log.h>
```

```
/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>
#define RST_MAX_GET_ROBJS_DELAY 3        /* max seconds between polls */


/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * Local function prototypes
 */

NEW_SRC_FILE();
```

```
/***************************************************************************
 *
 * EDMRST_SetPrevBackup API
 *
 * Function Description:
 *       Set the restore_context to that of the previous backup with respect
 *       to the current one.
 *
 * Parameters:
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *  flags  (I) - Selection Flags: e.g., Complete backups only/partial ok
 *
 * Return Codes:
 *       E_SUCCESS       - operation completed successfully
 *       EP_RB_RECOVER_RPC_FAIL  - if comms with restore engine fail
 *       EP_RB_NO_PREV_CATALOG   - when at the first catalog
 *       EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the file
 *                                         of the new catalog
 *
 ******************************************************************/

eerrno_ty
EDMRST_SetPrevBackup( serverHandle       svrHdl,
                      u_long             flags )
{
        eerrno_ty               result;
        RE_set_backup_time_args         rpc_args;
        RE_status_result        *rpc_result;
        RE_status_result        *rpc_result_1;
        RE_null_args            null_args;

        rbe_log_debug_sub( 0, "EDMRST_SetPrevBackup called" );

        /* validate args first: */
        if (svrHdl==NULL)
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.flags = flags;
        set_rpc_obj( re_set_prev_backup, &rpc_args.RPCobjID );

        rpc_result = re_set_prev_backup_1( &rpc_args, svrHdl );

        if (NULL == rpc_result)
          {
            result = EP_RB_RECOVER_RPC_FAIL;
            rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
          }
        else
          {
            if (E_SUCCESS != rpc_result->status) {
              result = EP_RB_RECOVER_RPC_FAIL;
              rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
            }
            else
              {
                result = EP_RB_RECOVER_RPC_INCOMPLETE;

                /*        rpc_args_.maxEntries = maxEntries;*/

                /* poll for completion or error */
                while (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                  {
                    unsigned int poll_delay = 100000;   /* .1 second */
                    set_rpc_obj(
                        re_set_previous_backup_result, &null_args.RPCobjID );
```

```
                    rpc_result_1 = re_set_previous_backup_result_1(
                                                      &null_args, svrHdl);

                    if (!rpc_result_1)
                      {
                        result = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                      }
                    else
                      {
                        result = rpc_result_1->status;
                      }

                    if (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                      {
                        /* release RPC result struct: contents and struct */
                        xdr_free( xdr_RE_status_result,
                                  (char *)rpc_result_1 );
                        rpc_result_1 = NULL;
                        /* wait till next poll */
                        usleep( poll_delay );
                        if (poll_delay < RST_MAX_GET_ROBJS_DELAY) {
                          poll_delay *= 2;
                          if (poll_delay > RST_MAX_GET_ROBJS_DELAY)
                            poll_delay = RST_MAX_GET_ROBJS_DELAY;
                        }
                      }
                  }
              }
          }

        if (rpc_result_1 != NULL)
          {
            /* release RPC result struct: contents and struct */
            xdr_free (xdr_RE_status_result, (char *)rpc_result_1);
          }
        return result;
}
/* EDMRST_SetPrevBackup */

/***************************************************************************
 *
 * EDMRST_SetNextBackup API
 *
 * Function Description:
 *  This routine sets the recover environment to the the next backup
 *  of the specified work item.
 *
 * Parameters:
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *  flags  (I) - Selection Flags: e.g., Complete backups only/partial ok
 *
 * Return Codes:
 *       E_SUCCESS       - operation completed successfully
 *       EP_RB_RECOVER_RPC_FAIL  - if comms with restore engine fail
 *       EP_RB_NO_NEXT_CATALOG - when at the most recent catalog
 *       EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the file
 *                                         of the new catalog
 *       EP_RB_RECOVER_NO_CATALOG - when mcat_set_mcplane failed
 *
 ******************************************************************/

eerrno_ty
EDMRST_SetNextBackup( serverHandle       svrHdl,
                      u_long             flags )
```

```
{
        eerrno_ty                result;
        RE_set_backup_time_args          rpc_args;
        RE_status_result         *rpc_result;
        RE_status_result         *rpc_result_1;
        RE_null_args             null_args;

        rbe_log_debug_sub( 0, "EDMRST_SetNextBackup called" );

        /* validate args first: */
        if (svrHdl==NULL)
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.flags = flags;
        set_rpc_obj( re_set_next_backup, &rpc_args.RPCobjID );

        rpc_result = re_set_next_backup_1( &rpc_args, svrHdl );

        if (NULL == rpc_result)
          {
            result = EP_RB_RECOVER_RPC_FAIL;
            rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
          }
        else
          {
            if (E_SUCCESS != rpc_result->status) {
              result = EP_RB_RECOVER_RPC_FAIL;
              rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
            }
            else
              {
                result = EP_RB_RECOVER_RPC_INCOMPLETE;

                /*        rpc_args_.maxEntries = maxEntries;*/

                /* poll for completion or error */
                while (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                  {
                    unsigned int poll_delay = 100000;   /* .1 second */
                    set_rpc_obj(
                            re_set_next_backup_result, &null_args.RPCobjID );
                    rpc_result_1 = re_set_next_backup_result_1(
                                                     &null_args, svrHdl);

                    if (!rpc_result_1)
                      {
                        result = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                      }
                    else
                      {
                        result = rpc_result_1->status;
                      }

                    if (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                      {
                        /* release RPC result struct: contents and struct */
                        xdr_free( xdr_RE_status_result,
                                  (char *)rpc_result_1 );
                        rpc_result_1 = NULL;
                        /* wait till next poll */
                        usleep( poll_delay );
                        if (poll_delay < RST_MAX_GET_ROBJS_DELAY) {
                          poll_delay *= 2;
                          if (poll_delay > RST_MAX_GET_ROBJS_DELAY)
                            poll_delay = RST_MAX_GET_ROBJS_DELAY;
```

```
                        }
                      }
                  }
              }
          }

        if (rpc_result_1 != NULL)
          {
            /* release RPC result struct: contents and struct */
            xdr_free (xdr_RE_status_result, (char *)rpc_result_1);
          }
        return result;

}
/* EDMRST_SetNextBackup */


/*************************************************************************
 *
 * EDMRST_SetFirstBackup API
 *
 * Function Description:
 *       Set the recover_context to that of the first backup catalog plane.
 *
 * Parameters:
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *  flags  (I) - Selection Flags: e.g., Complete backups only/partial ok
 *
 * Return Codes:
 *       E_SUCCESS       - operation completed successfully
 *
 *       EP_RB_RECOVER_RPC_FAIL  - if comms with restore engine fail
 *       EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the file of
 *                                         the new catalog
 *
 ********************************************************************/

eerrno_ty
EDMRST_SetFirstBackup( serverHandle      svrHdl,
                       u_long            flags )
{
        eerrno_ty                result;
        RE_set_backup_time_args          rpc_args;
        RE_status_result         *rpc_result;
        RE_status_result         *rpc_result_1;
        RE_null_args             null_args;

        rbe_log_debug_sub( 0, "EDMRST_SetFirstBackup called" );

        /* validate args first: */
        if (svrHdl==NULL)
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.flags = flags;
        set_rpc_obj( re_set_first_backup, &rpc_args.RPCobjID );

        rpc_result = re_set_first_backup_1( &rpc_args, svrHdl );

        if (NULL == rpc_result)
          {
            result = EP_RB_RECOVER_RPC_FAIL;
            rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
          }
        else
          {
            if (E_SUCCESS != rpc_result->status) {
              result = EP_RB_RECOVER_RPC_FAIL;
              rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
            }
            else
              {
                result = EP_RB_RECOVER_RPC_INCOMPLETE;

                /*        rpc_args_.maxEntries = maxEntries;*/

                /* poll for completion or error */
                while (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                  {
                    unsigned int poll_delay = 100000;   /* .1 second */
                    set_rpc_obj(
                          re_set_first_backup_result, &null_args.RPCobjID );
                    rpc_result_1 = re_set_first_backup_result_1(
                                                       &null_args, svrHdl);

                    if (!rpc_result_1)
                      {
                        result = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                      }
                    else
                      {
                        result = rpc_result_1->status;
                      }

                    if (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                      {
                        /* release RPC result struct: contents and struct */
                        xdr_free( xdr_RE_status_result,
                                   (char *)rpc_result_1 );
                        rpc_result_1 = NULL;
                        /* wait till next poll */
                        usleep( poll_delay );
                        if (poll_delay < RST_MAX_GET_ROBJS_DELAY) {
                          poll_delay *= 2;
                          if (poll_delay > RST_MAX_GET_ROBJS_DELAY)
                            poll_delay = RST_MAX_GET_ROBJS_DELAY;
                        }
                      }
                  }
              }
          }

        if (rpc_result_1 != NULL)
          {
            /* release RPC result struct: contents and struct */
            xdr_free (xdr_RE_status_result, (char *)rpc_result_1);
          }
        return result;

}
/* EDMRST_SetFirstBackup */


/**********************************************************************
 *
 * EDMRST_SetBackupForTime API
 *
 * Function Description:
 *      Set the recover_context to that of the backup catalog plane of the
 *      specified time.
 *
```

```
 * Parameters:
 *  svrHdl   (I) - A pointer to this user's client handle for the
 *                 Restore Engine (server) connection.
 *  forTime (I) - The time for which the backup is requested
 *  flags    (I) - Selection Flags: e.g., Complete backups only/partial ok
 *
 * Return Codes:
 *       E_SUCCESS       - operation completed successfully
 *       EP_RB_RECOVER_RPC_FAIL  - if comms with restore engine fail
 *       EP_RB_RECOVER_NO_CATALOG        - catalog cannot be found
 *
 **********************************************************************/

eerrno_ty
EDMRST_SetBackupForTime( serverHandle svrHdl,
                         const time_t forTime,
                         u_long       flags )
{
        eerrno_ty              result;
        RE_backup_for_time_args        rpc_args;
        RE_status_result       *rpc_result;
        RE_status_result       *rpc_result_1;
        RE_null_args           null_args;

        rbe_log_debug_sub( 0, "EDMRST_SetBackupForTime called" );

        /* validate args first: */
        if (svrHdl==NULL)
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.time = forTime;
        rpc_args.flags = flags;
        set_rpc_obj( re_set_backup_for_time, &rpc_args.RPCobjID );

        rpc_result = re_set_backup_for_time_1( &rpc_args, svrHdl );

        if (NULL == rpc_result)
          {
            result = EP_RB_RECOVER_RPC_FAIL;
            rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
          }
        else
          {
            if (E_SUCCESS != rpc_result->status) {
              result = EP_RB_RECOVER_RPC_FAIL;
              rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
            }
            else
              {
                result = EP_RB_RECOVER_RPC_INCOMPLETE;

                /*        rpc_args_.maxEntries = maxEntries;*/

                /* poll for completion or error */
                while (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                  {
                    unsigned int poll_delay = 100000;   /* .1 second */
                    set_rpc_obj(
                        re_set_backup_for_time_result, &null_args.RPCobjID );
                    rpc_result_1 = re_set_backup_for_time_result_1(
                                                       &null_args, svrHdl);

                    if (!rpc_result_1)
                      {
                        result = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                    }
                  else
                    {
                      result = rpc_result_1->status;
                    }

                  if (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                    {
                      /* release RPC result struct: contents and struct */
                      xdr_free( xdr_RE_status_result,
                                (char *)rpc_result_1 );
                      rpc_result_1 = NULL;
                      /* wait till next poll */
                      usleep( poll_delay );
                      if (poll_delay < RST_MAX_GET_ROBJS_DELAY) {
                        poll_delay *= 2;
                        if (poll_delay > RST_MAX_GET_ROBJS_DELAY)
                          poll_delay = RST_MAX_GET_ROBJS_DELAY;
                      }
                    }
                }
            }
        }

        if (rpc_result_1 != NULL)
          {
            /* release RPC result struct: contents and struct */
            xdr_free (xdr_RE_status_result, (char *)rpc_result_1);
          }
        return result;

}       /* EDMRST_SetBackupForTime */

/*********************************************************************
 *
 * EDMRST_GetCurrentBackupTime
 *
 * Function Description:
 *       Retrieve the time of the backup that the current recover_context
 *       is set to and return it in the preallocated buffer.
 *
 * Parameters:
 *       svrHdl -    (I) A pointer to this user's client handle for the
 *                       Restore Engine (server) connection.
 *       bkupTime - (O) the time of the backup
 *
 * Return Codes:
 *       E_SUCCESS        - operation completed successfully
 *       EP_RB_RECOVER_RPC_FAIL  - if comms with restore engine fail
 *       EP_RB_RECOVER_INVALOP   - call issued out of sequence
 *       EP_RB_RECOVER_BAD_ARGS  - invalid input argument
 *       EP_RB_RECOVER_NO_CURR_BACKUP    - no valid backup currently
 *
 *****************************************************************/

eerrno_ty
EDMRST_GetCurrentBackupTime( serverHandle svrHdl,
                             time_t        *bkupTime )
{
        eerrno_ty       result;
        RE_null_args    rpc_args;
        RE_get_current_backup_time_result *rpc_result;

        rbe_log_debug_sub( 0, "EDMRST_GetCurrentBackupTime called" );

        /* validate args first: */
```

```
        if (svrHdl==NULL || bkupTime==NULL)
                return( EP_RB_RECOVER_BAD_ARGS );

        set_rpc_obj( re_get_current_backup_time, &rpc_args.RPCobjID );

        rpc_result = re_get_current_backup_time_1( &rpc_args, svrHdl );
        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                result = rpc_result->status;
                *bkupTime = rpc_result->backupTime;

                /* release RPC result struct: contents and struct */
                xdr_free( xdr_RE_get_current_backup_time_result,
                          (char *)rpc_result );
        }

        return result;
}       /* EDMRST_GetCurrentBackupTime */

/**********************************************************************
 *
 * EDMRST_SetMostRecentBackup API
 *
 * Function Description:
 *       Set the recover_context to that of the most recent backup catalog
 *       plane. The recover_context will be set accordingly.
 *
 * Parameters:
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *  flags  (I) - Selection Flags: e.g., Complete backups only/partial ok
 *
 * Return Codes:
 *       E_SUCCESS        - operation completed successfully
 *
 *       EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the file of
 *                                         the new catalog.
 *
 ******************************************************/

eerrno_ty
EDMRST_SetMostRecentBackup( serverHandle svrHdl,
                            u_long       flags )
{
        eerrno_ty               result;
        RE_set_backup_time_args         rpc_args;
        RE_status_result        *rpc_result;
        RE_null_args            null_args;
        RE_status_result        *rpc_result_1;

        rbe_log_debug_sub( 0, "EDMRST_SetMostRecentBackup called" );

        /* validate args first: */
        if (svrHdl==NULL)
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.flags = flags;
        set_rpc_obj( re_set_most_recent_backup, &rpc_args.RPCobjID );

        rpc_result = re_set_most_recent_backup_1( &rpc_args, svrHdl );

        if (NULL == rpc_result)
```

```
          {
            result = EP_RB_RECOVER_RPC_FAIL;
            rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
          }
        else
          {
            if (E_SUCCESS != rpc_result->status) {
              result = EP_RB_RECOVER_RPC_FAIL;
              rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
            }
            else
              {
                result = EP_RB_RECOVER_RPC_INCOMPLETE;

                /*        rpc_args_.maxEntries = maxEntries;*/

                /* poll for completion or error */
                while (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                  {
                    unsigned int poll_delay = 100000;   /* .1 second */
                    set_rpc_obj(
                     re_set_most_recent_backup_result, &null_args.RPCobjID );
                    rpc_result_1 = re_set_most_recent_backup_result_1(
                                                    &null_args, svrHdl);

                    if (!rpc_result_1)
                      {
                        result = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                      }
                    else
                      {
                        result = rpc_result_1->status;
                      }

                    if (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                      {
                        /* release RPC result struct: contents and struct */
                        xdr_free( xdr_RE_status_result,
                                  (char *)rpc_result_1 );
                        rpc_result_1 = NULL;
                        /* wait till next poll */
                        usleep( poll_delay );
                        if (poll_delay < RST_MAX_GET_ROBJS_DELAY) {
                          poll_delay *= 2;
                          if (poll_delay > RST_MAX_GET_ROBJS_DELAY)
                            poll_delay = RST_MAX_GET_ROBJS_DELAY;
                        }
                      }
                  }
              }
          }

        if (rpc_result_1 != NULL)
          {
            /* release RPC result struct: contents and struct */
            xdr_free (xdr_RE_status_result, (char *)rpc_result_1);
          }
        return result;

}
/* EDMRST_SetMostRecentBackup */


/**********************************************************************
 *
```

```
 * EDMRST_GetAllBackupTimes API
 *
 * Function Description:
 *       Retrieve the dates of the backups within the time range
 *       specified by the caller.
 *
 *  The cookie must be initialize to INIT_COOKIE on the first call to this
 *  routine. This routine will update the cookie to allow retrieval of more
 *  objects if there is more than "maxEntries".  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * Parameters:
 *       svrHdl          - (I) A pointer to this user's client handle for the
 *                             Restore Engine (server) connection.
 *       startTime       - (I) include no earlier than this date
 *       endTime                 - (I) include no later than this date
 *       flags           - (
                         I) Backup constraint flags: e.g. full-only/partial-ok
 *       maxEntries      - (I) size of the array timesArray
 *       timesArray      - (I) ptr to array of time_t buffers
 *       numEntries      - (O) count of times returned
 *       cookie          - (IO) marker to specify whether or not this is
 *                              the initial call
 *
 * Return Codes:
 *       E_SUCCESS       - operation completed successfully
 *
 */

eerrno_ty
EDMRST_GetAllBackupTimes( serverHandle svrHdl,
                          const time_t startTime,
                          const time_t endTime,
                          u_long flags,
                          const short maxEntries,
                          time_t *timesArray,
                          short *numEntries,
                          long *cookie )
{
        eerrno_ty                         result;
        RE_get_all_backup_times_args      rpc_args;
        RE_get_all_backup_times_result    *rpc_result_1=NULL;
        RE_status_result                  *rpc_result;
        int                               indx;
        RSTRPC_time_list                  *lnkPtr;
        RE_null_args                      null_args;
        rbe_log_debug_sub( 0, "EDMRST_GetAllBackupTimes called" );

        /* validate args first: */
        if (
         timesArray==NULL || svrHdl==NULL || numEntries==NULL || cookie==NULL
          || maxEntries <= 0)
              return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.startTime = startTime;
        rpc_args.endTime = endTime;
        rpc_args.flags = flags;
        rpc_args.maxEntries = maxEntries;
        rpc_args.cookie = *cookie;
        set_rpc_obj( re_get_all_backup_times, &rpc_args.RPCobjID );

        rpc_result = re_get_all_backup_times_1( &rpc_args, svrHdl );
        /*pointer issues*/
        if (NULL == rpc_result)
          {
            result = EP_RB_RECOVER_RPC_FAIL;
            rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
          }
        else
          {
            if (E_SUCCESS != rpc_result->status) {
              result = EP_RB_RECOVER_RPC_FAIL;
              rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
            }
            else
              {
                result = EP_RB_RECOVER_RPC_INCOMPLETE;

                /*        rpc_args_.maxEntries = maxEntries;*/

                /* poll for completion or error */
                while (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                  {
                    unsigned int poll_delay = 100000;   /* .1 second */
                    set_rpc_obj(
                       re_get_all_backup_times_result, &null_args.RPCobjID );
                    rpc_result_1 = re_get_all_backup_times_result_1(
                                                    &null_args, svrHdl);

                    if (!rpc_result_1)
                      {
                        result = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
                      }
                    else
                      {
                        result = rpc_result_1->status;
                      }

                    if (result == EP_RB_RECOVER_RPC_INCOMPLETE)
                      {
                        /* release RPC result struct: contents and struct */
                        xdr_free( xdr_RE_get_all_backup_times_result,
                                  (char *)rpc_result_1 );
                        rpc_result_1 = NULL;
                        /* wait till next poll */
                        usleep( poll_delay );
                        if (poll_delay < RST_MAX_GET_ROBJS_DELAY) {
                          poll_delay *= 2;
                          if (poll_delay > RST_MAX_GET_ROBJS_DELAY)
                            poll_delay = RST_MAX_GET_ROBJS_DELAY;
                        }
                      }
                  }


            if ( E_SUCCESS == result)
              {
                *numEntries = rpc_result_1->numEntries;
                *cookie = rpc_result_1->cookie;
                lnkPtr = rpc_result_1->backupTimes;

                for( indx=0;
                     (indx < maxEntries) && (
                                   indx < rpc_result_1->numEntries );
                     indx++, timesArray++ )
                  {
                    /*
                     * If we are in the for loop but the linked list ptr
                     * is NULL,
                           then some internal inconsistency has occurred.
```

```
                     */
                    if (NULL == lnkPtr)
                      {
                        result = EP_RB_RECOVER_RPC_FAIL;
                        *numEntries = indx;
                        break;
                      }

                    *timesArray = lnkPtr->time;
                    lnkPtr = lnkPtr->next;
                  }

              }
          }
      }
    if (rpc_result_1 != NULL)
      {
        /* release RPC result struct: contents and struct */
        xdr_free (xdr_RE_get_all_backup_times_result, (
                                                      char *)rpc_result_1);
      }
    return result;
}
        /* EDMRST_GetAllBackupTimes */


/************************************************************************
 *
 * EDMRST_GetCurrentTemplate API
 *
 * Function Description:
 *  This routine returns the name of the template that is used by
 *      the currently selected top level object (work item) and the flag
 *      on whether or not the alternate trail is being used.
 *
 * Parameters:
 *      svrHdl          - (I) A pointer to this user's client handle for the
 *                            Restore Engine (server) connection.
 *      template        - (O) ptr to a preallocated template_name_ty
 *                            buffer
 *      alternate       - (O) ptr to a preallocate boolean_ty var
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *      EP_RB_RECOVER_BAD_ARGS  - invalid input argument
 *      EP_RB_RECOVER_NO_CURR_TEMPLATE  - no valid current template
 *
 */

eerrno_ty
EDMRST_GetCurrentTemplate( serverHandle     svrHdl,
                           template_name_ty template,
                            boolean_ty       *alternate )
{
        eerrno_ty                        result;
        RE_null_args                     rpc_args;
        RE_get_current_template_result  *rpc_result;

        rbe_log_debug_sub( 0, "EDMRST_GetCurrentTemplate called" );

        if (NULL == svrHdl)
                return EP_RB_RECOVER_INVALOP ;

        if ((NULL == template) || (NULL == alternate))
                return EP_RB_RECOVER_BAD_ARGS ;
	set_rpc_obj( re_get_current_template, &rpc_args.RPCobjID );

	rpc_result = re_get_current_template_1( &rpc_args, svrHdl );
	if (!rpc_result) {
		result = EP_RB_RECOVER_RPC_FAIL;
		rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
	}
	else
	{
		result = rpc_result->status;
		*alternate = rpc_result->alternate;
		strncpy(
		      template, rpc_result->templateName, MAX_TEMPLATE_LEN );

		/* release RPC result struct: contents and struct */
		xdr_free (xdr_RE_get_current_template_result, (
							 char *)rpc_result);
	}

	return result;
}
	/* EDMRST_GetCurrentTemplate */
```

```
/***************************************************************************
**
** File Name:   RSTgtshost.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**       This module contains:
**              -EDMRST_GetSourceHosts: The Restore API function, which
**               retrieves the hosts which are restorable by a given user.
**              -EDMRST_GetBackupServers:  The Restore API function which
**               retrieves the server hosts which have this host configured
**               for backup.
**
**
** Compile-Time Options:
**             This section must list any compile time definitions
**             which will affect this header.
**
***************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  __EXTENSIONS__   /* instead of _POSIX_SOURCE because of gethostname
                                                                           */


/*
 * System headers.
 */
#include <sys/param.h>   /* for MAXHOSTNAMELEN */
#include <unistd.h>      /* for gethostname */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <restore/restore_engine.h>
#include <RSTsup_rpc.h>
#include <RSTsup_csm.h>
```

```
/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * External declarations
 */
```

```
/*********************************************************************
 * EDMRST_GetSourceHosts:
 *
 *  This function is provided to allow retrieval of the
 *  hosts which are restorable by a given user.
 *
 *  Goal:
 *    For a host to be restorable there must have been at least one
 *    successful backup.
 *
 *  The cookie must be initialized to INIT_COOKIE on the first call to this
 *  routine. This routine will update the cookie to allow retrieval of more
 *  objects if there are more than "maxEntries".  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * Parameters:
 *   svrHdl        (I)  - A pointer to this user's client handle for the
 *                         Restore Engine (server) connection.
 *   hostname      (I)  - If NULL, its a no-op. Otherwise, the list of
 *                         recoverable hosts will be filtered based on
 *                         the value of "hostname".
 *   maxEntries    (I)  - the maximum number of hosts to return
 *   hosts         (O)  - a pre-allocated array to return the hosts in
 *   numberEntries (O)  - the real number of hosts returned in the array
 *   cookie        (IO) - a place holder for the list position
 *                          meaningful to only the internals of the API
 *
 ******************************************************************/

eerrno_ty
EDMRST_GetSourceHosts(   serverHandle svrHdl,
                         const char   *hostname,
                         const short  maxEntries,
                         char         **hosts,
                         short        *numberEntries,
                         long         *cookie )
{
        RE_get_hosts_result     *rpc_result;
        RE_get_hosts_args       rpc_args;
        RSTRPC_name_list        *temp_list;
        eerrno_ty               result;

        /* validate args first: */
        if (svrHdl==NULL || hosts==NULL || numberEntries==NULL
          || cookie==NULL || maxEntries <= 0 )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* Prepare input argument structure for RPC: */
        rpc_args.hostname = (char *)hostname;
        rpc_args.maxEntries = maxEntries;
        rpc_args.cookie = *cookie;
        set_rpc_obj( re_get_source_hosts, &rpc_args.RPCobjID );

        rpc_result = re_get_source_hosts_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }

        /* move results to caller's area, if successful: */
        else
        {
                result = rpc_result->status;
                if (rpc_result->status == E_SUCCESS)
                  {
```

```
                    *cookie = rpc_result->cookie;
                    *numberEntries = rpc_result->numEntries;
                    temp_list = rpc_result->hosts;
                    while ( rpc_result->numEntries )
                      {
                        /* check for  null pointer or too many returned */
                        if (!temp_list || !*hosts || !rpc_args.maxEntries--
                            || !temp_list->name )
                          break;
                        strcpy( *hosts++, temp_list->name );
                        temp_list = temp_list->next;
                        rpc_result->numEntries--;
                      }
                    if (rpc_result->numEntries)
                      result = EP_RB_RECOVER_SERVERFAIL;
                  }

                /* release RPC result struct: */
                xdr_free( xdr_RE_get_hosts_result, (char *)rpc_result );
        }

        return( result );

}        /* End of EDMRST_GetSourceHosts() */

/*********************************************************************
 * EDMRST_GetBackupServers:
 *
 *  This function is provided to allow retrieval, one at a time, of the
 *  servers which are configured to backup (and restore) this host.
 *
 *  The cookie must be initialized to INIT_COOKIE on the first call to this
 *  routine. This routine will update the cookie to allow retrieval of more
 *  server names if there are more than one.  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * ********************  NOTE: ***********************************
 * In this implementation, the restore gui can only run on the EDM server,
 * so only the current host can be the backup server.
                                                When other configurations
 * are possible, i.e., multiple EDM servers are possible,
                                                    this function must be
 * updated to determine the possible servers.  Presumably,
                                                    this will be through
 * a call to the Dispatch Daemon, to get the list of EDM servers.  Then those
 * servers can be queried to see if the current (local) host is one of its
 * backup clients.
 *********************************************************************
 *
 * Parameters:
 *   hostname      (O)  - Pointer to buffer to receive the server name output
 *   cookie        (IO) - a place holder for the list position
 *                          meaningful to only the internals of the API
 *
 *********************************************************************/

eerrno_ty
EDMRST_GetBackupServers( hostname_ty hostname,
                         long        *cookie )
{
        int     status;
        static long     valid_cookie = INIT_COOKIE;

        if (NULL == hostname || NULL == cookie)
              return EP_RB_RECOVER_BAD_ARGS;
        if (*cookie == INIT_COOKIE) {
                status = gethostname( hostname, MAXHOSTNAMELEN );
                if (status)
                        return EP_RB_RECOVER_FATALERR;
                *cookie = valid_cookie = DONE_COOKIE;
        }
        else if (*cookie == DONE_COOKIE || *cookie != valid_cookie)
                return EP_RB_RECOVER_BAD_COOKIE;
        else {  /* cant happen yet */
                 return EP_RB_RECOVER_FATALERR;
        }

        return E_SUCCESS;
}
        /* End of EDMRST_GetBackupServers() */
```

```
/**************************************************************************
 **
 ** File Name:   RSTgtdhost.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **              This module contains the Get Destination Hosts
 **              Restore API function.
 **
 ** Table of Contents:
 ** ------------------
 **       API Functions:
 **              EDMRST_GetDestinationHosts
 **
 **       Internal Functions:
 **
 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
 *************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif

#define  _POSIX_SOURCE 1

/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */


/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <restore/restore_engine.h>
#include <RSTsup_rpc.h>
#include <RSTsup_csm.h>
```

```
/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * External declarations
 */
```

```
/*****************************************************************
 * Get Destination Hosts:
 *
 *  This function is provided to allow retrieval of the
 *  hosts which are allowable destinations for the source host
 *  by a given user.
 *
 *  The cookie must be initialize to INIT_COOKIE on the first call to this
 *  routine. This routine will update the cookie to allow retrieval of more
 *  objects if there is more than "maxEntries".  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * Parameters:
 *   svrHdl        (I)  - A pointer to this user's client handle for the
 *                        Restore Engine (server) connection.
 *   maxEntries    (I)  - the maximum number of hosts to return
 *   hosts         (O)  - a pre-allocated array to return the hosts in
 *   numberEntries (O)  - the real number of hosts returned in the array
 *   cookie        (IO) - a place holder for the list position
 *
 *****************************************************************/

eerrno_ty
EDMRST_GetDestinationHosts( serverHandle svrHdl,
                            const short  maxEntries,
                            hostname_ty  *hosts,
                            short        *numberEntries,
                            long         *cookie )
{
        RE_get_hosts_result     *rpc_result;
        RE_get_hosts_args       rpc_args;
        RSTRPC_name_list        *temp_list;
        eerrno_ty               result;

        /* validate args first: */
        if (svrHdl==NULL || hosts==NULL || numberEntries==NULL
          || cookie==NULL || maxEntries <= 0 )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* Prepare input argument structure for RPC: */
        rpc_args.hostname = NULL;
        rpc_args.maxEntries = maxEntries;
        rpc_args.cookie = *cookie;
        set_rpc_obj( re_get_destination_hosts, &rpc_args.RPCobjID );

        rpc_result = re_get_destination_hosts_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }

        /* move results to caller's area, if successful: */
        else if (rpc_result->status == E_SUCCESS)
        {
                result = rpc_result->status;
                *cookie = rpc_result->cookie;
                *numberEntries = rpc_result->numEntries;
                temp_list = rpc_result->hosts;
                while ( rpc_result->numEntries )
                {
                        /* check for  null pointer or too many returned */
                        if (!temp_list || !*hosts || !rpc_args.maxEntries--
                          || !temp_list->name )
                                break;
                        strcpy( *hosts++, temp_list->name );
```

```
                        temp_list = temp_list->next;
                        rpc_result->numEntries--;
                }
                if (rpc_result->numEntries)
                        result = EP_RB_RECOVER_SERVERFAIL;

                /* release RPC result struct: */
                xdr_free( xdr_RE_get_hosts_result, (char *)rpc_result );
        }

        return result;
}
        /* End of EDMRST_GetDestinationHosts() */
```

```
/*************************************************************************
 **
 ** File Name:   RSTmarkunm.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **               This module contains the Restore API functions to mark and
 **               unmark objects for restoral.
 **
 ** Table of Contents:
 ** -------------------
 **       API Functions:
 **               EDMRST_MarkObject
 **               EDMRST_GetMarkResults
 **               EDMRST_UnmarkObject
 **               EDMRST_GetUnmarkResults
 **               EDMRST_GetMarkedTotalSize
 **
 **       Internal Functions:
 **
 ** Compile-Time Options:
 **
 **
 ** NOTE: Part of this module is adapted from:
 ** server/libs/recover/grandfathered/cmd_markunmark.c
 ** It contains mainly support routines needed by the mark and unmark
 ** API functions.
 **
 ************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
```

```
/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>


/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * External declarations
 */


NEW_SRC_FILE();

/*
 * Local function prototypes
 */


/*******************************************************************
 * EDMRST_MarkObject()
 *   This function is passed a restorableObject and begins to mark files for
 *   restoral based on the input criteria.  A second function,
 *   EDMRST_GetMarkResults, is used to test for completion of the mark.
 *
 * Parameters:
 *
 *  svrHdl     (I) - A pointer to this user's client handle for the
 *                   Restore Engine (server) connection.
 *  thisObject (I) - The input object to mark, must be of type file or
 *                    directory (not a top level object) .
 *    1) For files thisObject is the represents the target file to be marked.
 *    2) For directories thisObject represents the directory to be recovered
 *    and if the descend parameter is true then the mark applies to all the
 *    contents of the directory.
 *    3) For Witems thisObject an error condition will be returned.
 *
 *  time       (I) - (optional) the backup time to perform the mark on --
 *                   if not specified, uses currrently selected backup; if
 *                   specified, leaves selected backup time unchanged
 *  allowBadfiles (I) - allows marking of files of state BADDATA.
 *  descend    (I) - Should mark operation descend to operate on the contents
 *                    of directories.
 *
 *******************************************************************/

eerrno_ty
EDMRST_MarkObject( serverHandle          svrHdl,
                   restorableObjectPtr   thisObject,
                   time_t                time,
                    boolean_ty           allowBadFiles,
                    boolean_ty           descend )
{
        RE_mark_object_result           *rpc_result;
        RE_mark_object_args             rpc_args;
        RSTRPC_restorable_obj_root      *temp_robj;
        eerrno_ty                       result = E_SUCCESS ;

        rbe_log_debug_sub( 0, "EDMRST_MarkObject called" );
        /* validate args first: */
        if (thisObject==NULL || svrHdl==NULL )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* validate input object type as RESTORABLE_OBJECT */
        temp_robj = ((restorableObject *)thisObject)->rpcObjPtr;
        if ( NULL == temp_robj
          || RESTORABLE_OBJECT !=
                        ((restorableObject *)thisObject)->restoreObjType )
                return EP_RB_RECOVER_INVAL_OBJTYPE;

        /* validate input object type as NOT top level */
        if ( (temp_robj->objLevel != RSTRPC_leaf_type)
          && (temp_robj->objLevel != RSTRPC_container_type) )
        {
                if (temp_robj->objLevel != RSTRPC_tlo_type)
                        return( EP_RB_RECOVER_INVAL_OBJTYPE );
                else
                        return( EP_RB_RECOVER_INVALOP );
        }

        /* Prepare input argument structure for RPC: */
        rpc_args.thisObj = (RSTRPC_user_restorable_object *)temp_robj;
        rpc_args.allowBadFiles = allowBadFiles;
        rpc_args.descend = descend;
        rpc_args.backupTime = time;
        set_rpc_obj( re_mark_object, &rpc_args.RPCobjID );

        rpc_result = re_mark_object_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
               rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                result = rpc_result->status;
               /* release RPC result struct: */
               xdr_free( xdr_RE_mark_object_result, (char *)rpc_result);
        }

        return( result );

}       /* end of EDMRST_MarkObject () */
```

```
/*********************************************************************
 * EDMRST_GetMarkResults()
 *
 *      This function tests for completion and retrieves the results of the
 * previously started mark operation.
 *
 * Parameters:
 *
 *  svrHdl      (I) - A pointer to this user's client handle for the
 *                      Restore Engine (server) connection.
 *
 *  interrupt     (I) - requests cancellation of the mark (if TRUE)
 *                       WARNING: If the operation is aborted, the mark will be
 *                       left in an unknown state.  That is, any one of the
 *                       descendants of the marked object may be marked or not.
 *                       It is up to the caller to determine how to proceed
 *                       afterwards.
 *  BadFilesCount (
 *                 O) -- returns the file count of bitfiles marked with BADDATA
 *  PermDenyFilesCount (O) -- returns the file count with permission denied
 *                 bitfiles that were not marked.
 *
 *  fileMarked (O) -- return the total files marked after this mark occurred.
 *  dirMarked (O) -- return the total directories marked after this mark
 *                   occurred.
 *  otherMarked (O) -- return the total "other" files marked after this mark.
 *
 ********************************************************************/

eerrno_ty
EDMRST_GetMarkResults(   serverHandle     svrHdl,
                        boolean_ty       interrupt,
                         u_long           *BadFilesCount,
                         u_long           *PermDenyFilesCount,
                         u_long           *fileMarked,
                         u_long           *dirMarked,
                         u_long           *otherMarked )
{
        RE_get_mark_results_result      *rpc_result;
        RE_get_mark_results_args        rpc_args;
        eerrno_ty                       result = E_SUCCESS ;

        rbe_log_debug_sub( 0, "EDMRST_GetMarkResults called" );

        /* validate args first: */
        if ( svrHdl==NULL || BadFilesCount==NULL
          || fileMarked==NULL || PermDenyFilesCount==NULL
          || dirMarked==NULL || otherMarked==NULL
        )
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.interrupt = interrupt;
        set_rpc_obj( re_get_mark_results, &rpc_args.RPCobjID );
        rpc_result = re_get_mark_results_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
               rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {  /* move results to caller's area, if successful: */
                result = rpc_result->status;
                if (result == E_SUCCESS)
                {
                        *BadFilesCount = rpc_result->badFileCount;
                        *PermDenyFilesCount = rpc_result->permDenyFileCount;
                        *dirMarked = rpc_result->dirMarkCount;
                        *fileMarked = rpc_result->fileMarkCount;
                        *otherMarked = rpc_result->otherMarkCount;
                }

                /* release RPC result struct: */
                xdr_free( xdr_RE_get_mark_results_result, (
                                                        char *)rpc_result );
        }

        return( result );
}

/***************************************************************
 * UnmarkObject() and GetUnmarkResults()
 *
 *   UnmarkObject operates like MarkObject, in that it is supported through
 *   two API calls -- UnmarkObject and GetUnmarkresults.  Unmark starts an
 *   asynchronous operation in the Restore Engine to perform the unmarking,
 *   and returns.
 *
 *   GetUnmarkResults is called to test for completion of the unmark operation,
 *   and receive results when it is done.  It can also be used to interrupt
 *   the unmark operation.
 *
 * UnmarkObject Parameters:
 *
 *  svrHdl        (I) - A pointer to this user's client handle for the
 *                      Restore Engine (server) connection.
 *  thisObject    (I) - The restoral object; can be a leaf object (e.g. a
 *                      file), or a container object (e.g., a directory).
 *  backupTime    (I) - (optional) the backup time to perform the unmark on --
 *                      if not specified, uses currrently selected backup; if
 *                      specified, leaves selected backup time unchanged
 *  BadFilesOnly  (I) - allows unmarking ONLY of files of state BADDATA.
 *  descend       (I) - Should unmark operation descend to operate on the
 *                      content of container objects.
 ***************************************************************/

eerrno_ty EDMRST_UnmarkObject( serverHandle        svrHdl,
                               restorableObjectPtr thisObject,
                               time_t              backupTime,
                               boolean_ty          BadFilesOnly,
                               boolean_ty          descend )
{
        RE_mark_object_result           *rpc_result;
        RE_unmark_object_args           rpc_args;
        RSTRPC_restorable_obj_root      *temp_robj;
        eerrno_ty                       result = E_SUCCESS ;

        rbe_log_debug_sub( 0, "EDMRST_UnmarkObject called" );

        /* validate args first: */
        if (thisObject==NULL || svrHdl==NULL )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* validate input object type as RESTORABLE_OBJECT */
        temp_robj = ((restorableObject *)thisObject)->rpcObjPtr;
        if (NULL == temp_robj || RESTORABLE_OBJECT !=
                        ((restorableObject *)thisObject)->restoreObjType )
                return EP_RB_RECOVER_INVAL_OBJTYPE;

        /* validate input object type as NOT top level */
        if ( (temp_robj->objLevel != RSTRPC_leaf_type)
          && (temp_robj->objLevel != RSTRPC_container_type) )
        {
                if (temp_robj->objLevel != RSTRPC_tlo_type)
```

```
                        return( EP_RB_RECOVER_INVAL_OBJTYPE );
                else
                        return( EP_RB_RECOVER_INVALOP );
        }

        /* Prepare input argument structure for RPC: */
        rpc_args.thisObj = (RSTRPC_user_restorable_object *)temp_robj;
        rpc_args.badFilesOnly = BadFilesOnly;
        rpc_args.descend = descend;
        rpc_args.backupTime = backupTime;
        set_rpc_obj( re_unmark_object, &rpc_args.RPCobjID );

        rpc_result = re_unmark_object_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
               rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                result = rpc_result->status;
               /* release RPC result struct: */
               xdr_free( xdr_RE_mark_object_result, (char *)rpc_result);
        }

        return( result );
}
/* end of EDMRST_UnmarkObject () */

/***************************************************************
 *
 * GetUnmarkResults Parameters:
 *
 *  svrHdl        (I) - A pointer to this user's client handle for the
 *                      Restore Engine (server) connection.
 *  interrupt     (I) - requests cancellation of the unmark (if TRUE)
 *                      WARNING: If the operation is aborted, the unmark will
 *                      be left in an unknown state.  That is, any one of the
 *                      descendants of the unmarked object may be marked or
 *                      not. It is up to the caller to determine how to
 *                      proceed afterwards.
 *  BadFilesCount (O) - returns the file count with BADDATA.
 *  fileMarked    (
                  O) - return the total files marked after this mark occurred.
 *  dirMarked     (O) - return the total directories marked after this mark
 *                      occurred.
 *  otherMarked   (O) - return the total "other" files marked after this mark.
 *
 ***************************************************************/

eerrno_ty EDMRST_GetUnmarkResults( serverHandle svrHdl,
                                   boolean_ty   interrupt,
                                   u_long       *BadFilesCount,
                                   u_long       *fileMarked,
                                   u_long       *dirMarked,
                                   u_long       *otherMarked )
{
        RE_get_unmark_results_result    *rpc_result;
        RE_get_mark_results_args        rpc_args;
        eerrno_ty                       result = E_SUCCESS ;

        rbe_log_debug_sub( 0, "EDMRST_GetUnmarkResults called" );

        /* validate args first: */
        if ( svrHdl==NULL || BadFilesCount==NULL || fileMarked==NULL
          || dirMarked==NULL || otherMarked==NULL
        )
```

```
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.interrupt = interrupt;
        set_rpc_obj( re_get_unmark_results, &rpc_args.RPCobjID );
        rpc_result = re_get_unmark_results_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
               rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {  /* move results to caller's area, if successful: */
                result = rpc_result->status;
                if (result == E_SUCCESS)
                {
                        *BadFilesCount = rpc_result->badFileCount;
                        *dirMarked = rpc_result->dirMarkCount;
                        *fileMarked = rpc_result->fileMarkCount;
                        *otherMarked = rpc_result->otherMarkCount;
                }

                /* release RPC result struct: */
                xdr_free( xdr_RE_get_unmark_results_result, (
                                                char *)rpc_result );
        }

        return( result );
}

/**********************************************
 * EDMRST_GetMarkedTotalSize ()
 *
 * This function is provided to allow retrieval of the
 * Total size of the marked files.
 *
 * size is the sum-of-the-length the marked files and is one
 * measure of size. This is an approximation.
 *
 *********************************************/

eerrno_ty
EDMRST_GetMarkedTotalSize( serverHandle svrHdl,
                           u_hyper      *totalSize )
{
        RE_get_marked_total_size_result         *rpc_result;
        RE_null_args                    rpc_args;
        eerrno_ty                       result = E_SUCCESS ;

        rbe_log_debug_sub( 0, "EDMRST_GetMarkedTotalSize called" );

        /* validate args first: */
        if ( svrHdl==NULL || totalSize==NULL )
                return( EP_RB_RECOVER_BAD_ARGS );

        set_rpc_obj( re_get_marked_total_size, &rpc_args.RPCobjID );
        rpc_result = re_get_marked_total_size_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
               rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {  /* move results to caller's area, if successful: */
                result = rpc_result->status;
                if (result == E_SUCCESS)
                {
                        totalSize->high = rpc_result->total.high;
                        totalSize->low = rpc_result->total.low;
```

```
                }

                /* release RPC result struct: */
                xdr_free( xdr_RE_get_marked_total_size_result,
                        (char *)rpc_result );
        }

        return( result );
}
/*      EDMRST_GetMarkedTotalSize */
```

```
/*************************************************************************
 **
 ** File Name:   RSTmedia.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **       This module contains the Restore API functions that provide
 **       the information of the media needed for restore access. This
 **       media list is updated in EDMRST_MarkObject()
 **       and EDMRST_UnmarkObject().
 **
 ** Table of Contents:
 ** --------------------
 ** public functions contained in:
 **       EDMRST_GetNecessaryMedia
 **       EDMRST_GetMediaVolid
 **       EDMRST_GetMediaLabel
 **       EDMRST_GetMediaSide
 **      EDMRST_GetMediaSequenceNumber
 **       EDMRST_GetMediaBarcodeString
 **      EDMRST_GetMediaDescription
 **       EDMRST_GetMediaStatus
 **       EDMRST_GetMediaTrail
 **      EDMRST_GetMediaLocation
 **      EDMRST_GetMediaComments
 **
 **       EDMRST_GetNumberOfDuplicates
 **      EDMRST_GetDupilcateVolid
 **      EDMRST_GetDupilcateSequenceNumber
 **       EDMRST_GetDupilcateBarcodeString
 **      EDMRST_GetDupilcateTypeDescription
 **      EDMRST_GetDupilcateTypeToken
 **       EDMRST_GetDupilcateStatus
 **       EDMRST_GetDupilcatetTrail
 **      EDMRST_GetDupilcateLocation
 **
 ** static functions NO LONGER contained here:
 **       InitializeMediaObjects
 **       AssignMediaObjects
 **      MediaObjectConstructor
 **       validateMediaObject
 **       volid2str
 **
 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
*************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
```

```
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>


/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * External declarations
 */


NEW_SRC_FILE();

/*
 * Local function prototypes
 */

static eerrno_ty CheckMediaObjects( const short maxEntries,
                                    mediaObject **objects );

static eerrno_ty copy_rpc_media_obj( mediaObject *dest,
                                     RSTRPC_media_object *src );
static eerrno_ty copy_rpc_media_dups( mediaObject *dest,
                                      RSTRPC_media_object *src );
```

```
/* public functions */

/*****************************************************************
 * Get Necessary Media:
 *
 *  This function is provided to allow retrieval of the
 *  necessary media to restore the currently marked objects.
 *
 *  The cookie must be initialize to INIT_COOKIE  on the first call to
 *  this routine. This routine will update the cookie to allow retrieval
 *  of more objects if there is more than "maxEntries".  The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * Parameters:
 *   svrHdl        (I)- a pointer to this user's client handle for the
 *                      Restore Engine (server) connection.
 *   maxEntries    (I)- the maximum number of media objects to return
 *   objects       (O)- an allocated array to return the objects in
 *   numberEntries (O)- the real number of media objects returned in the array
 *   cookie       (IO)- a place holder for the list position
 *
 *****************************************************************/

eerrno_ty
EDMRST_GetNecessaryMedia( serverHandle   svrHdl,
                          const short    maxEntries,
                          mediaObjectPtr *objects,
                          short          *numberEntries,
                          boolean_ty     all,
                          long           *cookie )
{
        RE_get_necessary_media_result   *rpc_result;
        RE_get_necessary_media_args     rpc_args;
        RSTRPC_media_list               *temp_list;
        eerrno_ty                       result = E_SUCCESS;

        /* validate inputs; */
        if (NULL == svrHdl || NULL == numberEntries || NULL == cookie ||
           maxEntries <= 0 )
                return EP_RB_RECOVER_BAD_ARGS;

        if (E_SUCCESS != (result = CheckMediaObjects( maxEntries,
                                        (mediaObject **)objects)))
        {
            return result;
        }

        /* Prepare input argument structure for RPC: */
        rpc_args.maxEntries = maxEntries;
        rpc_args.cookie = *cookie;
        rpc_args.all = all;
        set_rpc_obj( re_get_necessary_media, &rpc_args.RPCobjID );

        rpc_result = re_get_necessary_media_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                if ( (result = rpc_result->status) == E_SUCCESS)
                {
                    *cookie = rpc_result->cookie;
                    *numberEntries = rpc_result->numEntries;
                    temp_list = rpc_result->mediaList;
                    while ( rpc_result->numEntries && result == E_SUCCESS )
```

```
                    {
                        if (!temp_list || !*objects || !rpc_args.maxEntries--
                         || !temp_list->media_obj )
                                break;  /* some null pointer or too many */
                        /* copy list object to array object entry: */
                         result = copy_rpc_media_obj( *objects,
                                                  temp_list->media_obj );
                         /* copy the duplicates for EACH media object
                          * into the media list stored in each original
                          * media object
                          */
                         result = copy_rpc_media_dups( *objects++,
                                                  temp_list->media_obj );
                        temp_list = temp_list->next;
                        rpc_result->numEntries--;
                    }
                }

                /* release RPC result struct: */
                 xdr_free( xdr_RE_get_necessary_media_result,
                        (char *)rpc_result );
        }

        return result;
}
        /* EDMRST_GetNecessaryMedia */

static eerrno_ty
CheckMediaObjects(const short maxEntries,
                  mediaObject **objects)
{
    register int index;


    if (NULL == objects)
    {
        return EP_RB_RECOVER_BAD_ARGS;
    }

    for (index = 0; index < maxEntries; index++)
    {
        if (NULL == objects[index]
         || MEDIA_OBJECT != objects[index]->restoreObjType)
        {
            return EP_RB_RECOVER_BAD_ARGS;
        }
    }
    return E_SUCCESS;
}       /* CheckMediaObjects */


static eerrno_ty copy_rpc_media_obj( mediaObject *dest,
                                     RSTRPC_media_object *src )
{
        if ( NULL == (dest->trail = esl_strdup( src->trail )) )
        {
                rec_api_log_csm(SUB_CSM_NOMEM, NULL);
                return EP_RB_RECOVER_NOMEM;
        }
        if ( NULL == (dest->mtype = esl_strdup( src->mtype )) )
        {
                rec_api_log_csm(SUB_CSM_NOMEM, NULL);
                return EP_RB_RECOVER_NOMEM;
        }
        if ( NULL == (dest->mtype_token = esl_strdup( src->mtype_token )) )
        {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
            return EP_RB_RECOVER_NOMEM;
        }
        if ( NULL == (dest->barcode_label = esl_strdup(
                                          src->barcode_label )) )
        {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
            return EP_RB_RECOVER_NOMEM;
        }
        if ( NULL == (dest->physical_loc = esl_strdup( src->physical_loc )) )
        {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
            return EP_RB_RECOVER_NOMEM;
        }
        if ( NULL == (dest->comments = esl_strdup( src->comments )) )
        {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
            return EP_RB_RECOVER_NOMEM;
        }
        if ( NULL == (dest->volid_ascii = esl_strdup( src->volid_ascii )) )
        {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
            return EP_RB_RECOVER_NOMEM;
        }
        if ( NULL == (dest->luname = esl_strdup( src->luname )) )
        {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
            return EP_RB_RECOVER_NOMEM;
        }
        dest->seqno = src->seqno;
        dest->side = src->side;
        dest->lmtime = src->lmtime;
        dest->online = src->online;
        dest->offsite = src->offsite;
        dest->is_orig = src->is_orig;
        dest->run_media_dup = src->run_media_dup;
        /* added this to copy the number of duplicates */
        dest->num_dups = src->num_dups;
        return E_SUCCESS;
}


/*****************************************************
 * copy_rpc_media_dups
 * goes through the list of duplicates from the RSTRPC_media_object
 * and calls the copy media obj function to copy the fields into
 * the new structure.  This just copies the linked list of duplicates
 *****************************************************
 */
static eerrno_ty
copy_rpc_media_dups( mediaObject *dest,
                     struct RSTRPC_media_object *src )
{

struct mediaObjectList    *dst_list_pointer;
                                /* The list of objects to be copied to */
 struct RSTRPC_media_list *src_list_pointer;
                                  /* List of objects to be copied from */
eerrno_ty              result=E_SUCCESS;

 dest->dups=calloc(1,sizeof(struct mediaObjectList));
                                     /* creates the first list item  */

dst_list_pointer = dest->dups;
```

```
src_list_pointer = src->dups;

/* traverse the source media list */
while(src_list_pointer!=NULL)
  {
    dst_list_pointer->media_obj = calloc(1,sizeof(mediaObject));
    result = copy_rpc_media_obj(
                    dst_list_pointer->media_obj, src_list_pointer->media_obj);
    if (result != E_SUCCESS)  /* if the copy had an error lets exit */
      return result;

    src_list_pointer=src_list_pointer->next;
                                        /* move to the next media object */

    if (src_list_pointer != NULL)  /* if we still have more to copy */
      {
        dst_list_pointer->next=calloc(1,sizeof(struct mediaObjectList));
        dst_list_pointer=dst_list_pointer->next;
      }
    else  /* no more to copy */
      {
        dst_list_pointer->next=NULL;
      }
  }
return result;
}

/*****************************************************************
 * Media Object Access Routines:
 *
 *  These routines retrieve information pertinent to a given Media object.
 *
 * Parameters:
 *
 *  svrHdl      (I) - (ignored) A pointer to this user's client handle for the
 *                    Restore Engine (server) connection.
 *  thisObject (I) - The media object
 * For the duplicate functions
 *  dup_number (I) - The number of the duplicate to retrieve from usually
 *                   1 for now, until multiple duplicates can be made
 *
 * RETURNS one of the following:
 *       const char *    pointer to a string within the media object,
                                                                 that should
 *                       not be changed.
 *       MediaStatus
 *       long            media sequence number
 *       uchar_t                 media side
 *
 ****************************************************************/

const char *
EDMRST_GetMediaVolid( serverHandle   svrHdl,
                      mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
     || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
     || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->volid_ascii;
}       /* EDMRST_GetMediaVolid */

const char *
EDMRST_GetMediaLUName( serverHandle   svrHdl,
                        mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->luname;
}       /* EDMRST_GetMediaVolid */

const char *
EDMRST_GetMediaTrail( serverHandle   svrHdl,
                      mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->trail;
}       /* EDMRST_GetMediaTrail */

uchar_t
EDMRST_GetMediaSide( serverHandle   svrHdl,
                     mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return 0;

    return ((mediaObject *)thisObject)->side;
}       /* EDMRST_GetMediaSide */

long
EDMRST_GetMediaSequenceNumber( serverHandle   svrHdl,
                               mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return 0;

    return (long)((mediaObject *)thisObject)->seqno;
}       /* EDMRST_GetMediaSequenceNumber */

const char *
EDMRST_GetMediaBarcodeString( serverHandle   svrHdl,
                              mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->barcode_label;
}       /* EDMRST_GetMediaBarcodeString */

const char *
EDMRST_GetMediaTypeDescrip( serverHandle   svrHdl,
                            mediaObjectPtr thisObject )
```

```
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->mtype;
}       /* EDMRST_GetMediaTypeDescrip */

const char *
EDMRST_GetMediaTypeToken( serverHandle   svrHdl,
                          mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->mtype_token;
}       /* EDMRST_GetMediaTypeToken */

MediaStatus
EDMRST_GetMediaStatus( serverHandle   svrHdl,
                       mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return Media_Offline;

    if (((mediaObject *)thisObject)->online)
    {
        return Media_Online;
    }
    else if (!(((mediaObject *)thisObject)->offsite))
    {
        /*
         * offline & onsite
         */

        return Media_Offline;
    }
    else
    {
        /*
         * offsite & offline
         */

        return Media_Offsite;
    }
}       /* EDMRST_GetMediaStatus */

const char *
EDMRST_GetMediaLocation( serverHandle   svrHdl,
                         mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
      || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->physical_loc;
```

```
}        /* EDMRST_GetMediaLocation */

const char *
EDMRST_GetMediaComments( serverHandle   svrHdl,
                         mediaObjectPtr thisObject )
{
    if ( (NULL == svrHdl) || (NULL == thisObject)
     || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
     || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
    )
        return NULL;

    return ((mediaObject *)thisObject)->comments;
}        /* EDMRST_GetMediaComments */


/****************************************************
 * Duplicate Media Access Routines
 * Inputs: Svr Handle - see above
 *         dup_number: the number of the duplicate wanted
 *         thiObject:  The media object to get the dups for...
 ****************************************************
 */
short EDMRST_GetNumberOfDuplicates( serverHandle   svrHdl,
                                    mediaObjectPtr thisObject )
{
  mediaObject *tempObj;
  tempObj = (mediaObject *)thisObject;
  return tempObj->num_dups;
}



const char *
EDMRST_GetDuplicateVolid( serverHandle   svrHdl,
                          int dup_number,
                          mediaObjectPtr thisObject )
{
  mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                          * this is the original, but
                                          * the variable becomes the
                                          * duplicate  further down */
  dupObjectList = dupObject->dups;  /* Points to First mediaObject */

  /* Make sure we have something to work with */
  if ( (NULL == svrHdl) || (NULL == thisObject)
       || (dup_number >dupObject->num_dups)
       || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
       || (MEDIA_OBJECT != (dupObject->restoreObjType)))
    return NULL;

  /* get to specified Object, but already at first one */
  for (curr_dup=1; curr_dup<dup_number; curr_dup++)
    {
      dupObjectList = dupObjectList->next;
    }

  /* get the media object */
  dupObject = (mediaObject *) dupObjectList->media_obj;

  /* return the volid */
  return dupObject->volid_ascii;
```

```
        /* EDMRST_GetDuplicateVolid */
}



const char *
EDMRST_GetDuplicateLUName( serverHandle   svrHdl,
                           int dup_number,
                           mediaObjectPtr thisObject )
{
  mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                          * this is the original, but
                                          * the variable becomes the
                                          * duplicate  further down */
  dupObjectList = dupObject->dups;  /* Points to First mediaObject */

  /* Make sure we have something to work with */
  if ( (NULL == svrHdl) || (NULL == thisObject)
       || (dup_number >dupObject->num_dups)
       || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
       || (MEDIA_OBJECT != (dupObject->restoreObjType)))
    return NULL;

  /* get to specified Object, but already at first one */
  for (curr_dup=1; curr_dup<dup_number; curr_dup++)
    {
      dupObjectList = dupObjectList->next;
    }

  /* get the media object */
  dupObject = (mediaObject *) dupObjectList->media_obj;

  /* return the volid */
  return dupObject->luname;
        /* EDMRST_GetDuplicateVolid */
}

long
EDMRST_GetDuplicateSequenceNumber( serverHandle   svrHdl,
                                   int dup_number,
                                   mediaObjectPtr thisObject )
{
  mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                          * this is the original, but
                                          * the variable becomes the
                                          * duplicate  further down */
  dupObjectList = dupObject->dups; /* Points to First mediaObject */

  /* Make sure we have something to work with */
  if ( (NULL == svrHdl) || (NULL == thisObject)
       || (dup_number >dupObject->num_dups)
       || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
       || (MEDIA_OBJECT != (dupObject->restoreObjType)))
    return 0;

  /* get to specified Object, but already at first one */
  for (curr_dup=1;curr_dup<dup_number; curr_dup++)
    {
      dupObjectList = dupObjectList->next;
    }

  dupObject = (mediaObject *) dupObjectList->media_obj;

  return dupObject->seqno;
}         /* EDMRST_GetDuplicateSequenceNumber */




const char *
EDMRST_GetDuplicateBarcodeString( serverHandle   svrHdl,
                                  int dup_number,
                                  mediaObjectPtr thisObject )
{
  mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                          * this is the original, but
                                          * the variable becomes the
                                          * duplicate  further down */
  dupObjectList = dupObject->dups; /* Points to First mediaObject */

  /* Make sure we have something to work with */
  if ( (NULL == svrHdl) || (NULL == thisObject)
       || (dup_number >dupObject->num_dups)
       || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
       || (MEDIA_OBJECT != (dupObject->restoreObjType)))
    return NULL;

  /* get to specified Object, but already at first one */
  for (curr_dup=1;curr_dup<dup_number; curr_dup++)
    {
      dupObjectList = dupObjectList->next;
    }

  dupObject = (mediaObject *) dupObjectList->media_obj;

  return dupObject->barcode_label;
}         /* EDMRST_GetDuplicateBarcodeString */




const char *
EDMRST_GetDuplicateTypeDescrip( serverHandle   svrHdl,
                                int dup_number,
                                mediaObjectPtr thisObject )
{
   mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                          * this is the original, but
                                          * the variable becomes the
                                          * duplicate  further down */
  dupObjectList = dupObject->dups; /* Points to First mediaObject */

  /* Make sure we have something to work with */
  if ( (NULL == svrHdl) || (NULL == thisObject)
       || (dup_number >dupObject->num_dups)
```

```
       || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
       || (MEDIA_OBJECT != (dupObject->restoreObjType)))
    return NULL;

  /* get to specified Object, but already at first one */
  for (curr_dup=1;curr_dup<dup_number; curr_dup++)
    {
      dupObjectList = dupObjectList->next;
    }

  dupObject = (mediaObject *) dupObjectList->media_obj;

  return dupObject->mtype;
}         /* EDMRST_GetDuplicateTypeDescrip */




const char *
EDMRST_GetDuplicateTypeToken( serverHandle   svrHdl,
                              int dup_number,
                              mediaObjectPtr thisObject )
{
  mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                          * this is the original, but
                                          * the variable becomes the
                                          * duplicate  further down */
  dupObjectList = dupObject->dups; /* Points to First mediaObject */

  /* Make sure we have something to work with */
  if ( (NULL == svrHdl) || (NULL == thisObject)
       || (dup_number >dupObject->num_dups)
       || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
       || (MEDIA_OBJECT != (dupObject->restoreObjType)))
    return NULL;

  /* get to specified Object, but already at first one */
  for (curr_dup=1;curr_dup<dup_number; curr_dup++)
    {
      dupObjectList = dupObjectList->next;
    }

  dupObject = (mediaObject *) dupObjectList->media_obj;

  return dupObject->mtype_token;

}         /* EDMRST_GeDuplicatetTypeToken */




MediaStatus
EDMRST_GetDuplicateStatus( serverHandle   svrHdl,
                           int dup_number,
                           mediaObjectPtr thisObject )
{
  mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                               * this is the original, but
                                               * the variable becomes the
                                               * duplicate  further down */
   dupObjectList = dupObject->dups; /* Points to First mediaObject */

   /* Make sure we have something to work with */
   if ( (NULL == svrHdl) || (NULL == thisObject)
        || (dup_number >dupObject->num_dups)
        || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
          return Media_Offline;

   /* get to specified Object, but already at first one */
   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
     {
       dupObjectList = dupObjectList->next;
     }

  dupObject = (mediaObject *) dupObjectList->media_obj;

    if (dupObject->online)
    {
          return Media_Online;
    }
    else if (!(dupObject->offsite))
    {
          /*
           * offline & onsite
           */

          return Media_Offline;
    }
    else
    {
          /*
           * offsite & offline
           */

          return Media_Offsite;
    }
}         /* EDMRST_GetDuplicateStatus */




const char *
EDMRST_GetDuplicateLocation( serverHandle   svrHdl,
                              int dup_number,
                              mediaObjectPtr thisObject )
{
  mediaObject       *dupObject;
  struct mediaObjectList   *dupObjectList;
  short curr_dup=0;

  dupObject = (mediaObject *)thisObject; /* kinda cheating here,
                                          * this is the original, but
                                          * the variable becomes the
                                          * duplicate  further down */
  dupObjectList = dupObject->dups; /* Points to First mediaObject */

  /* Make sure we have something to work with */
  if ( (NULL == svrHdl) || (NULL == thisObject)
       || (dup_number >dupObject->num_dups)
       || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
       || (MEDIA_OBJECT != (dupObject->restoreObjType)))
```

```
     return NULL;

   /* get to specified Object, but already at first one */
   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
     {
       dupObjectList = dupObjectList->next;
     }

   dupObject = (mediaObject *) dupObjectList->media_obj;

   return dupObject->physical_loc;
}         /* EDMRST_GetDuplicateLocation */
```

```
/*************************************************************************
 **
 ** File Name:   RSTstart.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 ** -----------
 **       The intent of the contents of this file is to implement the
 **       functions the control execution of the restore for the Restore API.
 **
 **       These functions are provided to allow:
 **      - creation of submit objects, which define the set of objects to be
 **        restored and the scripts to be run before and after restoration,
 **      - starting the restoral of a submit object.
 **      - polling of the status of an ongoing restore,
                                                    including the ability to
 **        interrupt the restore, and to receive information necessary to
 **        query the user for input needed for the pre-restore or post-restore
 **        scripts, suspending, restarting,
 **      - supply of user responses to pre- and post- restore script queries
 **
 ** The following functions comprise restoral management:
 **
 **      EDMRST_Submit
 **      EDMRST_GetSubmitResults
 **      EDMRST_Start
 **      EDMRST_GetRestoreFeedback
 **      EDMRST_GetQuestion
 **      EDMRST_SetUserAnswer
 **
 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
*************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1
#define  NULL_STRING "\0"

/*
 * System headers.
 */
```

```
#include <sys/wait.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>



/*
 * #defines, structures, typedefs local to this source file
 */



/*
 * Command flags.
 */


/*
 * External declarations
 */


/*
 * Local function prototypes
 */
```

```
/*******************************************************************
 * Restoral Management Functions:
 *
 *  These functions are provided to allow:
 *      - creation of submit objects, which define the set of objects to be
 *        restored and the scripts to be run before and after restoration,
 *      - starting the restoral of a submit object.
 *      - polling of the status of an ongoing restore,
                                                   including the ability to
 *        interrupt the restore, and to receive information necessary to
 *        query the user for input needed for the pre-restore or post-restore
 *        scripts, suspending, restarting,
 *      - supply of user responses to pre- and post- restore script queries
 *
 * The following functions comprise restoral management:
 *
 *      EDMRST_Submit
 *      EDMRST_GetSubmitResults
 *      EDMRST_Start
 *      EDMRST_GetRestoreFeedback
 *       EDMRST_GetQuestion
 *      EDMRST_SetUserAnswer
 *      EDMRST_SetRecxDirectives
 *
 *******************************************************************
 * Submit
 *
 * This function starts the creation or update of a submit object from the
 * currently marked restorable objects. Its completion is tested for with
 * EDMRST_GetSubmitResults.  The returned submit object ID is passed to
 * EDMRST_Start to begin execution of the restore.
 *
 * Parameters:
 *
 * svrHdl        (I) - A pointer to this user's client handle for
 *                     the Restore Engine (server) connection.
 * policy        (I) - The overwrite policy to use
 * inPlace       (I) - Flag if the restoral is to be in original locations
 * hostName      (I) - host to restore to (only if inPlace == False)
 * directory     (I) - directory to restore to (only if inPlace == False)
 * transport     (I) - Indicator of transport the restoral is to be over (SCSI
 *                     or network)
 * submitObjID   (I) - ID of an existing submit object which is to be added to
 * socketClientNm (I) - Name of the client the restore is going to on
 *                      a client initiated restore
 * clientSocketPort (I) - the port to connect to on the client machine for
 * the restore
 *******************************************************************/
eerrno_ty EDMRST_Submit( serverHandle          svrHdl,
                         const char            *hostName,
                         const OverwritePolicy policy,
                         const boolean_ty      inPlace,
                         const char            *directory,
                         const RestoreTransport transport,
                         unsigned int          submitObjID,
                          EDMRST_submit_args    *submitArgs)
{
        RE_status_result        *rpc_result;
        RE_submit_args          rpc_args;
        eerrno_ty               result;
        char                    *nullstr = "";

        /* validate args first: */
        if ( svrHdl == NULL
         || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
          || (!inPlace && (hostName == NULL || directory == NULL) )
```

```
          )
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.overwritePolicy = policy;
        rpc_args.inPlace = inPlace;
        rpc_args.transport = transport;
        rpc_args.submitObjectID = submitObjID;

        if (NULL != submitArgs)
          {
            rpc_args.mapFile_env = esl_strdup(submitArgs->mapfile_env);
          }
        else
          {
            rpc_args.mapFile_env = nullstr;
          }

        if (NULL != submitArgs)
          {
            rpc_args.socketPort=submitArgs->clientSocketPort;
          }
        else
          {
            rpc_args.socketPort = 0;
          }

        if (NULL != submitArgs)
          {
            if (NULL==(rpc_args.socketClientName=esl_strdup(
                                            submitArgs->socketClientNm)))
              rpc_args.socketClientName=nullstr ;
          }
        else
          {
            rpc_args.socketClientName=nullstr ;
          }

        if (!inPlace) {
                rpc_args.hostname = (char *)hostName;
                rpc_args.directory = (char *)directory;
        } else {
                rpc_args.hostname = nullstr;
                rpc_args.directory = nullstr;
        }

        set_rpc_obj( re_submit, &rpc_args.RPCobjID );

        rpc_result = re_submit_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
        {
                result = rpc_result->status;

                /* release RPC result struct: contents and struct */
                xdr_free( xdr_RE_status_result, (char *)rpc_result );
        }
        return( result );
}

/*******************************************************************
 * GetSubmitResults
 *
 * This function tests for completion of an EDMRST_Submit call, with the
 * option of cancelling the submit.
 *
 * Parameters:
 *
 * svrHdl        (I) - A pointer to this user's client handle for
 *                     the Restore Engine (server) connection.
 * interrupt     (I) - Flag if the submit is to be canceled
 * submitObjID   (O) - ID of the submit object which describes the restore
 * objectsDone   (O) - number of objects -- total number in the submit object
 *                      if operation is complete,
 *                                           or number processed so far if
 *                      submit operation is still executing (
 *                                                     INCOMPLETE status)
 *
 ******************************************************************/
eerrno_ty EDMRST_GetSubmitResults( serverHandle     svrHdl,
                                   const boolean_ty interrupt,
                                   unsigned int     *submitObjID,
                                   unsigned long    *objectsDone )
{
        RE_get_submit_results_output    *rpc_result;
        RE_get_submit_results_args      rpc_args;
        eerrno_ty                       result;

        /* validate args first: */
        if (svrHdl == NULL || submitObjID == NULL || objectsDone == NULL
        || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
        )
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.interrupt = interrupt;

        set_rpc_obj( re_get_submit_results, &rpc_args.RPCobjID );

        rpc_result = re_get_submit_results_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
        {
                result = rpc_result->status;
                *objectsDone = rpc_result-> objectsDone;
                if (result == E_SUCCESS)
                        *submitObjID = rpc_result->submitObjectID;

                /* release RPC result struct: contents and struct */
                xdr_free (xdr_RE_get_submit_results_output, (
                                                   char *)rpc_result);
        }
        return( result );
}

/******************************************************************
 * Start
 *
 * This function begins execution of the restoral of the objects in a
 * submit object.  Its progress and requests for operator input are
 * received via EDMRST_GetRestoreFeedback.
 *
 * Parameters:
 *
 * svrHdl        (I) - A pointer to this user's client handle for
```

```
 *                     the Restore Engine (server) connection.
 * submitObjID   (I) - ID of the submit object that describes the restore
 *
 ******************************************************************/
eerrno_ty EDMRST_Start( serverHandle  svrHdl,
                        unsigned int submitObjID )
{
        RE_status_result        *rpc_result;
        RE_start_args           rpc_args;
        eerrno_ty               result;

        /* validate args first: */
        if (svrHdl == NULL || submitObjID == 0
        || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
        )
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.submitObjectID = submitObjID;

        set_rpc_obj( re_start, &rpc_args.RPCobjID );

        rpc_result = re_start_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
        {
                result = rpc_result->status;

                /* release RPC result struct: contents and struct */
                xdr_free( xdr_RE_status_result, (char *)rpc_result );
        }
        return( result );
}       /* EDMRST_Start */
```

```c
/*****************************************************************
 * GetRestoreFeedback
 *
 * This function is used to poll for the status of an ongoing restore, and
 * includes the ability to interrupt the restore, and to receive information
 * necessary to query the user for input needed for the pre-restore or
 * post-restore scripts.
 *
 * Parameters:
 *
 * svrHdl        (I) - A pointer to this user's client handle for
 *                     the Restore Engine (server) connection.
 * quitRestore   (I) - Flag if the restoral is to be stopped
 * currentState  (O) - Pointer to storage to receive the state of the restore
 * feedbackPtr  (IO) - Pointer to structure to receive restore feedback data
 *
 *****************************************************************/

eerrno_ty EDMRST_GetRestoreFeedback( serverHandle      svrHdl,
                                     const boolean_ty quitRestore,
                                     RERunningState   *currentState,
                                     feedbackObjectPtr feedbackPtr )
{
        RE_get_restore_feedback_result  *rpc_result;
        RE_get_restore_feedback_args    rpc_args;
        feedbackObject  *foPtr = (feedbackObject *)feedbackPtr;
        eerrno_ty               result;

        /* validate args first: */
        if (svrHdl == NULL || currentState == NULL || feedbackPtr == NULL
        || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
         || FEEDBACK_OBJECT != foPtr->restoreObjType
        )
                return( EP_RB_RECOVER_BAD_ARGS );

        FreeFeedbackObjectContents( foPtr );

        rpc_args.quit_restore = quitRestore;

        set_rpc_obj( re_get_restore_feedback, &rpc_args.RPCobjID );

        rpc_result = re_get_restore_feedback_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
        {
                result = rpc_result->status;
                foPtr->stats.status = rpc_result->rstStats.status;
                foPtr->stats.wiprogress = rpc_result->rstStats.wiprogress;
                rpc_result->rstStats.wiprogress = NULL;
                                                        /* avoid 2 frees */
                memcpy( &foPtr->stats.edm, &rpc_result->rstStats.edm,
                        sizeof (struct EDMProgress) );
                rpc_result->rstStats.edm.next = NULL;   /* for xdr_free */
                foPtr->notify = rpc_result->notify;
                rpc_result->notify = NULL;      /* avoid 2 frees */
                *currentState = rpc_result->rstStats.edm.status;

                /* release RPC result struct: contents and struct */
                xdr_free( xdr_RE_get_restore_feedback_result,
                          (char *)rpc_result );
        }
```

```c
        return( result );
}       /* EDMRST_GetRestoreFeedback */

/*****************************************************************
 * SetUserAnswer
 *
 * This function is used to return user input requested via the queryObjectPtr
 * parameter output of the EDMRST_GetQuestion function call.
 *
 * Parameters:
 *
 * svrHdl   (I) - A pointer to this user's client handle for
 *                the Restore Engine (server) connection.
 * queryPtr (I) - Pointer to object containing the question data.
 * answer   (I) - pointer to text string response to question
 * more     (I) - indicator that there will be more answers to this question
 *
 *****************************************************************/

eerrno_ty EDMRST_SetUserAnswer( serverHandle     svrHdl,
                                queryObjectPtr   queryPtr,
                                 const char      *answer,
                                 boolean_ty      more )
{
        RE_status_result         *rpc_result;
        RE_set_user_answer_args         rpc_args;
        Answer                   *answerObj, *tmpAnswer;
        queryObject              *queryObj = (queryObject *)queryPtr;
        eerrno_ty                result = E_SUCCESS;

        /* validate args first: */
        if (svrHdl == NULL || answer == NULL || queryPtr == NULL
        || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
         || (QUERY_OBJECT != queryObj->restoreObjType)
         || (NULL == queryObj->query)
        )
                return EP_RB_RECOVER_BAD_ARGS;

        /* allocate answer list if none in queryObject yet: */
        if (NULL == queryObj->answers)
        {
            if (NULL == (queryObj->answers = calloc( 1, sizeof(
                                                        AnswerList) )) )
                return EP_RB_RECOVER_NOMEM;
        }

        /* allocate and initialize answer object */
        if (NULL == (answerObj = calloc( 1, sizeof(Answer) )))
                return EP_RB_RECOVER_NOMEM;
        if (NULL == (answerObj->ctext = esl_strdup( answer )))
        {
                free( answerObj );
                return EP_RB_RECOVER_NOMEM;
        }
        answerObj->qnum = queryObj->query->qnum;
        answerObj->nextanswer = NULL;

        ++queryObj->answers->numanswers;
        if (NULL == (tmpAnswer = queryObj->answers->firstanswer))
                queryObj->answers->firstanswer = answerObj;
        else
        {
                while (NULL != tmpAnswer->nextanswer)
                        tmpAnswer = tmpAnswer->nextanswer;
                tmpAnswer->nextanswer = answerObj;
        }
        if (more)
                return result;

        /* prepare arg structures:  move answer list to rpc structure */

        rpc_args.answers.numanswers = queryObj->answers->numanswers;
        rpc_args.answers.firstanswer = queryObj->answers->firstanswer;
        queryObj->answers->firstanswer = NULL;
        set_rpc_obj( re_set_user_answer, &rpc_args.RPCobjID );

        rpc_result = re_set_user_answer_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
        {
                result = rpc_result->status;

                /* release RPC result struct: contents and struct */
                xdr_free( xdr_RE_status_result, (char *)rpc_result );
        }

        return( result );
}


/*****************************************************************
 * GetQuestion
 *
 * This function is used to fetch the data needed to query the user during a
 * pre-restore or post-restore script execution.
 *
 * Parameters:
 *
 * svrHdl   (I) - A pointer to this user's client handle for
 *                the Restore Engine (server) connection.
 * queryPtr (O) - Pointer to the object containing the question data.
 *
 *****************************************************************/

eerrno_ty EDMRST_GetQuestion( serverHandle      svrHdl,
                              queryObjectPtr    queryPtr )
{
        RE_get_question_result  *rpc_result;
        queryObject             *query_ptr = (queryObject *)queryPtr;
        RE_null_args            rpc_args;
        eerrno_ty               result;

        /* validate args first: */
        if (svrHdl == NULL || queryPtr == NULL
        || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
         || (QUERY_OBJECT != query_ptr->restoreObjType )
        )
                return( EP_RB_RECOVER_BAD_ARGS );

        /* free last question in query obj */
        FreeQueryObjectContents( query_ptr );

        set_rpc_obj( re_get_question, &rpc_args.RPCobjID );

        rpc_result = re_get_question_1( &rpc_args, svrHdl );
```

```
        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
        {
                result = rpc_result->status;
                query_ptr->query = rpc_result->query;
                                                        /* use returned obj */
                rpc_result->query = NULL;       /* avoid 2 frees */

                /* release RPC result struct: contents and struct */
                xdr_free( xdr_RE_get_question_result, (char *)rpc_result );
        }

        return( result );
}
        /* EDMRST_GetQuestion */

/*****************************************************************
 * SetRecxDirectives:
 *
 *  This routine returns sends the Filename and path plus hostname
 *    of the recx directives file, which was created by the command
 *    eb_dc_restore, to the server which then processes the recx
 *     directives
 *
 * Parameters:
 *  svrHdl    (I) - A pointer to this user's client handle for the
 *                    Restore Engine (server) connection.
 *  template  (O) - The name of the local recx file
 *  alternate (O) - the name of this host so the file can be tranfered
 *
 *****************************************************************/
eerrno_ty EDMRST_SetRecxDirectives( serverHandle     svrHdl,
                                     char              *filename,
                                    char              *hostname )
{

        RE_status_result        *rpc_result;
        RE_recx_file_info       rpc_args;
        RSTRPC_recx_file_info   fileinfo;
        eerrno_ty               result;
        char                    *nullstr = "";
        RE_status_result        *poll_result;
        RE_null_args            args;
        int                     count;

        /* validate args first: */
        if ( svrHdl == NULL
         || (NULL == handlePtr) || (svrHdl != handlePtr->re_binding_handle)
          || (hostname == NULL || filename == NULL || 0==strcmp(
                                  hostname,"")|| 0==strcmp(filename,"")) )
                return( EP_RB_RECOVER_BAD_ARGS );

        fileinfo.filename = esl_strdup(filename);
        fileinfo.hostname = esl_strdup(hostname);
        rpc_args.fileinfo = fileinfo;

        /*no object ID in file info structure*/
        set_rpc_obj( re_load_recx_directives, &rpc_args.RPCobjID );

        rpc_result = re_load_recx_directives_1( &rpc_args, svrHdl );

        if ((NULL==rpc_result) || (rpc_result->status != E_SUCCESS)) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
        {
              set_rpc_obj(
                      re_poll_load_recx_directives, &rpc_args.RPCobjID );

              /* Initialize the count and poll_result*/
              poll_result =  re_poll_load_recx_directives_1(&args, svrHdl );
              count = 0;
              /* check to see if the RPC is still running
                 until it finishes, or it is longer than 60 seconds.*/
              while((poll_result->status == EP_RB_RECOVER_RPC_INCOMPLETE)&&(
                                                            count <=60))
                {
                  poll_result =  re_poll_load_recx_directives_1(
                                                       &args, svrHdl );
                  sleep(1);
                  count++;
                }

              if (poll_result->status != E_SUCCESS)
                result = EP_RB_RECOVER_RPC_FAIL;
              else
                result = rpc_result->status;

              /* release RPC result struct: contents and struct */
              xdr_free( xdr_RE_status_result, (char *)rpc_result );
        }

        /*polling info stuf*/



        return( result );
}

/*******************************************************************
 * EDMRST_get_catalog_info:
 *
 *  This routine returns sends the fills the level string with the
 *  level for backup being restored
 *
 * Parameters:
 *  svrHdl          (I) - A pointer to this user's client handle for the
 *                          Restore Engine (server) connection.
 *  backup_time     (I) - Time of the backup that is beeing looked at
 *  *level          (O) - The level of the backup for specified time
 *                      taken from catalog structure.  If not enough
 *                      memory has been allocated value will be "\0"
 *  *numrec         (O) - The number of records for the specified backup
 *                      taken from catalog structure.   If not enough
 *                      memory has been allocated value will be "\0"
 *  *catType        (O) - The type of catalog for the specified bacup
 *                      taken from catalog structure.   If not enough
 *                      memory has been allocated value will be "\0"
 * Return Codes:
 *   EP_RB_RECOVER_BAD_ARGS - arguments passed in are null
 *   E_SUCCESS              - the fields have been filled in
 *                            and RPC succeeded
 *
 *******************************************************************/
eerrno_ty
EDMRST_getCatalogInfo( serverHandle   svrHdl,
                       time_t         backup_time,
```

```
                       char           *level,
                       char           *numrec,
                       char           *catType)
{

  RE_catalog_info               *rpc_result;
  RE_time                  rpc_args;
  char                     *result = NULL;
  int                      tmp;
  /* validate args first: */
  if ((0==backup_time) || (NULL==svrHdl ) || (NULL==level)
      || (NULL==numrec) || (NULL==catType))
          return(EP_RB_RECOVER_BAD_ARGS);

  /* Prepare input argument structure for RPC: */
  rpc_args.backupTime=backup_time;
  set_rpc_obj( re_get_catalog_info, &rpc_args.RPCobjID );

  rpc_result = re_get_catalog_info_1( &rpc_args, svrHdl );

  if (rpc_result == NULL)
    {
      rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
      return(EP_RB_RECOVER_RPC_FAIL);
    }

/*copy the structures level field to the level variable*/

 tmp = rpc_result->level[0];
 sprintf(level, "%d",tmp);
/*copy the structures numrec  field to the numrec variable*/
numrec=strcpy(numrec,rpc_result->numrec);
/*copy the structures catType field to the catType variable*/
catType=strcpy(catType,rpc_result->catType);

 return( E_SUCCESS );

}
```

```
/***************************************************************************
 **
 ** File Name:   RSTfind.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **       Implementation for EDMRST_FindRestorableObjects, which is the recover
 **       "find" command.  What is supported in find is what was supported in
 **       the old xebrecover find GUI.
 **
 ** Table of Contents:
 ** -------------------
 **       Restore API Functions:
 **               EDMRST_FindRestorableObjects
 **               EDMRST_GetFindResults
 **
 **       Internal Functions:
 **

 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
***************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif

#define  _POSIX_SOURCE 1


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */


/*
 * System headers.
 */

#include <grp.h>
#include <pwd.h>
#include <search.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
```

```
/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>


/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * External declarations
 */
```

```
/*******************************************************************
 * Find Routine:
 *
 *  These routines allow the user to find restorable objects.  Returned
 *  is an array of found objects and an array of backup times associated
 *  with the objects.  These arrays are 1-to-1.  That is, the nth Object
 *  was backed up at the nth time.
 *
 *  This operation is performed asynchronously by the Restore Engine, and the
 *  first API function, EDMRST_FindRestorableObjects, is used to start the
 *  'find'.  EDMRST_GetFindResults is used to test for completion of the find,
 *  cancel the find, and receive the results (
 *                                          parts of, at least) if it is done.
 *
 * EDMRST_FindRestorableObjects Parameters:
 *
 *   svrHdl         (I) - A pointer to this user's client handle for the
 *                        Restore Engine (server) connection.
 *   searchCriteria (I) - The criteria used for the search
 *
 *
 * Return:
 *       E_SUCCESS
 *       EP_EB_RECOVER_FIND_BAD_USER
 *       EP_EB_RECOVER_FIND_BAD_GROUP
 *       EP_RB_RECOVER_FIND_FAILED
 *       EP_RB_RECOVER_BAD_ARGS
 *       EP_RB_RECOVER_FIND_INTERRUPTED
 *       EP_RB_RECOVER_FATALERR
 *       others
 *
 *******************************************************************/

eerrno_ty EDMRST_FindRestorableObjects( serverHandle          svrHdl,
                                        EBREC_SearchCriteriaRec *searchCriteria
                                                                )

{
        RE_find_restorable_objects_result       *rpc_result;
        RE_find_restorable_objects_args                 rpc_args;
        RE_search_criteria                      criteria;
        eerrno_ty                           result = E_SUCCESS ;

        rbe_log_debug_sub( 0, "EDMRST_FindRestorableObjects called" );

        /* validate args first: */
        if (NULL == searchCriteria || NULL == svrHdl
          || NULL == searchCriteria->startDirectory)
                return( EP_RB_RECOVER_BAD_ARGS );

        /* Prepare input argument structure for RPC: */
        rpc_args.searchCriteria = &criteria;
        /* load criteria structure for RPC -- we dont dupe strings since they
           will only be used temporarily by the rpc call */
        criteria.startDirectory = searchCriteria->startDirectory;
        criteria.descendDirectory = searchCriteria->descendDirectory;
        criteria.searchString = searchCriteria->searchString;
        criteria.excludeString = searchCriteria->excludeString;
        criteria.typeOfFile = searchCriteria->typeOfFile;
        criteria.owner = searchCriteria->owner;
        criteria.excludeOwner = searchCriteria->excludeOwner;
        criteria.group = searchCriteria->group;
        criteria.excludeGroup = searchCriteria->excludeGroup;
        criteria.sizeInBytes.high = searchCriteria->sizeInBytes.high;
        criteria.sizeInBytes.low = searchCriteria->sizeInBytes.low;
        criteria.sizeMatch = searchCriteria->sizeMatch;
```

```
        criteria.startTime = searchCriteria->startTime;
        criteria.endTime = searchCriteria->endTime;

        set_rpc_obj( re_find_restorable_objects, &rpc_args.RPCobjID );

        rpc_result = re_find_restorable_objects_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                result = rpc_result->status;
                /* release RPC result struct: */
                xdr_free( xdr_RE_find_restorable_objects_result,
                          (char *)rpc_result);
        }

        return( result );

}       /* EDMRST_FindRestorableObjects */
```

```
/******************************************************************
 * GetFindResults
 *
 * EDMRST_GetFindResults is used to test for completion of the find,
 * and receive the results (parts of, at least) if it is done.
 *
 * Parameters:
 *
 *   svrHdl        (I)  - a pointer to this user's client handle for the
 *                        Restore Engine (server) connection.
 *   interrupt     (I)  - requests cancellation of the find (if TRUE)
 *   maxEntries    (I)  - the maximum number of found objects to return
 *   foundObjects  (O)  - a pre-allocated array to return the objects in
 *   times         (O)  - a pre-allocated array to return the backup times in
 *   numberEntries (O)  - the real number of objects returned in the array
 *   cookie        (IO) - a place holder for the list position
 *
 * Return:
 *       E_SUCCESS
 *       EP_RB_RECOVER_BAD_COOKIE
 *       EP_RB_RECOVER_BAD_ARGS
 *       others
 *
 ******************************************************************/
eerrno_ty EDMRST_GetFindResults( serverHandle         svrHdl,
                                 boolean_ty           interrupt,
                                 const long           maxEntries,
                                 restorableObjectPtr *foundObjects,
                                 time_t              *times,
                                 long                *numberEntries,
                                 long                *cookie )
{
        RE_get_find_results_result      *rpc_result = NULL;
        RE_get_find_results_args        rpc_args;
        RSTRPC_found_obj_list           *temp_list;
        eerrno_ty                       result = E_SUCCESS ;
        short                           index;
        restorableObject                **foundArray;

        rbe_log_debug_sub( 0, "EDMRST_GetFindResults called" );

        /* validate args first: */
        if (NULL == foundObjects || NULL == svrHdl || NULL == numberEntries
          || NULL == times || NULL == cookie || maxEntries <= 0)
                return( EP_RB_RECOVER_BAD_ARGS );

        /* validate target restorableObjects: */
        for ( foundArray=(restorableObject **)foundObjects, index=0;
              index<maxEntries;
              index++,foundArray++ )
        {
                if ( NULL == *foundArray
                  || RESTORABLE_OBJECT != (*foundArray)->restoreObjType
                  || NULL != (*foundArray)->rpcObjPtr )
                return( EP_RB_RECOVER_BAD_ARGS );
        }

        rpc_args.maxEntries = maxEntries;
        rpc_args.cookie = *cookie;
        rpc_args.interrupt = interrupt;

        /* call RPC, get response */
        set_rpc_obj( re_get_find_results, &rpc_args.RPCobjID );
        rpc_result = re_get_find_results_1( &rpc_args, svrHdl );
        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
```

```
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else
                result = rpc_result->status;

        /* move results to caller's area, if successful: */
        if (result == E_SUCCESS)
        {
                *cookie = rpc_result->cookie;
                *numberEntries = rpc_result->numEntries;
                index = 0;
                temp_list = rpc_result->foundObjs;
                foundArray = (restorableObject **)foundObjects;
                while ( rpc_result->numEntries )
                {
                        if ( !temp_list || !rpc_args.maxEntries-- )
                                break;
                                        /* null pointer or too many returned */
                        foundArray[index]->rpcObjPtr =
                            (RSTRPC_restorable_obj_root *)temp_list->foundObj;
                        times[index++] = temp_list->time;
                        temp_list->foundObj = NULL;
                /* needed to end with NULL in each RSTRPC_found_obj_list
                                                                        entry
                 * because returned user rest. objects can't be freed yet */
                        temp_list = temp_list->next;
                        rpc_result->numEntries--;
                }
                if (rpc_result->numEntries)     /* break'ed before done */
                        result = EP_RB_RECOVER_SERVERFAIL;
        }

        /* release RPC result struct's contents : */
        if (rpc_result) {
                xdr_free( xdr_RE_get_find_results_result, (
                                                        char *)rpc_result );
        }

        return( result );
}
        /* EDMRST_GetFindResults */

/******************************************************************/
```